SCADAPack² C++ Tools

User and Reference Manual

CONTROL MICROSYSTEMS
SCADA products... for the distance

48 Steacie Drive
Kanata, Ontario
K2K 2A9
Canada

Telephone: 613-591-1943
Facsimile: 613-591-1022
Technical Support: 888-226-6876
888-2CONTROL

SCADAPack^{2} C++ Tools User and Reference Manual

©2006 Control Microsystems Inc.
All rights reserved.

Printed in Canada.

Trademarks

TeleSAFE, TelePACE, TeleBUS, SmartWIRE and SCADAPack are registered trademarks of Control Microsystems Inc.

All other product names are copyright and registered trademarks or trade names of their respective owners.

Table of Contents

TABLE OF CONTENTS I

OVERVIEW 1

GETTING STARTED 2

C++ PROGRAM DEVELOPMENT 12

REAL TIME OPERATING SYSTEM 17

OVERVIEW OF PROGRAMMING FUNCTIONS 29

FUNCTION SPECIFICATIONS 53

MACRO DEFINITIONS 440

STRUCTURES AND TYPES 449

EXAMPLE PROGRAMS 489

PORTING EXISTING C TOOLS APPLICATIONS 524

Overview

The SCADAPack^2 C++ Tools are ideal for engineers and programmers who require advanced programming tools for SCADA applications and process control. The SCADAPack^2 controller executes TelePACE Ladder Logic or ISaGRAF and up to 32 C++ application programs simultaneously, providing you with maximum flexibility in implementing your control strategy.

This manual provides documentation on SCADAPack^2 C++ programming and the library of C++ language process control and SCADA functions. We strongly encourage you to read it, and to notify us if you find any errors or additional items you feel should be included in our documentation.

We sincerely hope that the reliability and flexibility afforded by this fully programmable controller enable you and your company to solve your automation problems in a cost effective and efficient manner.

Control Microsystems Technical Support

Support related to any part of this documentation can be directed to Control Microsystems Inc.

Users in Canada and the United States may call our technical support department free of charge. Telephone, facsimile and e-mail support is available from 8:00 to 18:30 (North American Eastern Time Zone) at the following numbers. When calling, please ask for a Technical Support Representative.

Telephone:	1-888-226-6876 (1-888-2CONTROL)
Facsimile:	1-613-591-3145
E-Mail:	technicalsupport@controlmicrosystems.com

Getting Started

This section of the C++ Tools User Manual describes the installation of C++ Tools and includes a Program Development Tutorial. The Program Development Tutorial leads the user through the steps involved in writing, compiling, linking and loading a C++ application program.

SCADAPack2 C++ Tools Installation

The SCADAPack C++ Tools install a gnu C++ compiler and SCADAPack^2 controller header and support files. Framework applications for TelePACE and ISaGRAF firmware are provided.

Any standard Editor may be used to create C++ applications.

TelePACE, ISaGRAF, or RealFLO applications are used to load applications into the SCADAPack^2 controller.

These installations are described in the following sections.

Installing SCADAPack2 C++ Tools

To install the SCADAPack^2 C++ Tools:

- Insert the SCADAPack^2 C++ Tools CD into your CD drive and follow the on-screen instructions.

The SCADAPack^2 C Tools is a command line compiler. Two system properties must be set for the compiler to work.

To modify system properties:

- From the Start menu or the Desktop, right click on **My Computer**.
- Select the **Advanced** tab.
- Click **Environment Variables**.
- In the *System Variables* section (at the bottom) add a variable as follows:
    - Click **New**.
    - In *Variable Name* type CTOOLS_PATH.
    - In *Variable Value* type C:\program files\Control Microsystems\CTools
      (if you installed to a different path, then substitute the correct path here)
    - Click **OK**.
- In the *System Variables* section (at the bottom) modify the PATH variable as follows:
    - Locate the PATH variable.
    - Click **Edit**.
    - In *Variable Value* add the following at the start of the text, including the semi-colon at the end of the string:
      C:\Program Files\Control Microsystems\CTools\Arm7\host\x86-win32\bin;
      (if you installed to a different path, then substitute the correct path here)
    - Click **OK**.

- Click **OK**.

Installing TelePACE

Install TelePACE as described on the jewel case liner of the TelePACE Installation CD.

Some virus checking software may interfere with Setup. If you experience problems with the Setup, disable your virus checker and run Setup again.

Installing ISaGRAF Workbench

Install ISaGRAF as described on the jewel case liner of the ISaGRAF Installation CD.

Some virus checking software may interfere with Setup. If you experience problems with the Setup, disable your virus checker and run Setup again.

Viewing Installed Components

The SCADAPack^2 C++ Tools installs the following components. All files are installed by default to C:\program files\Control Microsystems\CTools.

- gnu C++ compiler for Arm7 processor is installed in the ARM7 folder
- SCADAPack^2 C Tools header and support files are installed in:
    - SCADAPack2/ISaGRAF for ISaGRAF firmware applications
    - SCADAPack2/TelePACE for TelePACE firmware applications
- SCADAPack^2 Framework applications are installed in SCADAPack2/Framework Applications. These are described further in the product development tutorial.
- Documentation shortcuts are on the Start menu. You must have found them if you're reading this so we won't say any more.

Program Development Tutorial

Program development consists of three stages: writing and editing; compiling and linking; and loading the program into the target controller. Each step uses separate tools. To demonstrate these steps a sample program will be prepared.

Traditionally, the first program that is run on a new C compiler is the *hello, world* program. It prints the message "hello, world". Hey, who are we to be different?

Create a New C++ Application Framework

Any editor may be used to write and edit the application program for the SCADAPack2.

Copy SCADAPack2 C++ Application Framework

Begin by making a copy of the SCADAPack2 C++ application framework using the ISaGRAF sample application or the TelePACE sample application. By default the samples are installed at C:\program files\Control Microsystems\CTools\SCADAPack2\Framework Applications. Make a copy of either the ISaGRAF or TelePACE folder for your application.

For example:

- Copy files from `C:\program files\Control Microsystems\CTools\SCADAPack2\Framework Applications\ISaGRAF`.
- Copy files to `C:\projects\SP2\hello`

Review appstart.cpp

The appstart.cpp file defines the basic settings for the application, such as stack size, and main task priority. Most applications can use the settings in this file without modification.

Open `appstart.cpp` to review these application settings:

```
...
// Priority of the task main().
// Priority 100 is recommended for a continuously running task.
// A task with priority > 100 will never be given the CPU.
// See manual for details.
UINT32 mainPriority = 100;

// Stack space allocated to the task main().
// Note that at least 10 stack blocks are needed when calling fprintf().
UINT32 mainStack = 10;

// Application group assigned to the task main().
// A unique value is assigned by the system to the applicationGroup
// for this application. Use this variable in calls to create_task()
// by this application. See manual for details.
UINT32 applicationGroup = 0;
...
```

Edit main.cpp

For this tutorial the C code to print “hello world” to serial port 2 will be added to the main task. The “hello, world” message will be output to the *com2* serial port of the SCADAPack2 controller. A terminal connected to the port will display the message.

The `fprintf` function prints the message to the `com2` serial port.

Edit the `main.cpp` text and add the text shown in bold in the following section.

```
int main(void)
{
      // add program initialization here

      // Print the message
      fprintf(com2, "hello, world\r\n");

      // main loop
      while (TRUE)
      {
         // add remainder of program here
      }
}
```

Compiling the C++ Application

Once the editing of the project is completed the application needs to be compiled and linked. This produces an executable file that can be loaded into the SCADAPack2 controller.

Review makefile

The SCADAPack^2 C++ tools use the gnu make utility to build applications. Application builds are managed by a make file. For the simplest applications, no modifications of the makefile are needed. This section may be skimmed the first time through, but contains information that will be important for building more sophisticated applications.

Open the file `makefile` in the application folder. The file shown below is from the ISaGRAF application framework.

```
# ------------------------------------------------------------------------
# makefile
#
# Make file for SCADAPack2 C Tools application for ISaGRAF firmware
# Copyright 2006 Control Microsystems Inc.
#
# usage:
#    make       - makes the application
#    make clean - deletes all output files
# ------------------------------------------------------------------------
```

The first section of the file sets the name of the output file. The default name is `myApp`. You should modify this for you application.

```
# ------------------------------------------------------------------------
# set the name of the output file here
# ------------------------------------------------------------------------

APPLICATION_NAME = myApp
```

The next section lists all the object files in the application. There is one object file corresponding to each C or CPP source file. The framework has two files. You should add additional files here.

```
# ------------------------------------------------------------------------
# list all object files here
# ------------------------------------------------------------------------

objects = appstart.o main.o
```

The C Tools and include paths are set in the next section. The paths are taken from the environment variable you set during installation. If the variable is not present, they default to the standard paths. You don't need to do anything to this section.

```
# ------------------------------------------------------------------------
# set C Tools and include file paths
# ------------------------------------------------------------------------

# take the C Tools path from the environment, or set default if it's not there
(default may not be correct for all installations)
ifeq ($(strip $(CTOOLS_PATH)),)
CTOOLS_PATH = C:\Program Files\Control Microsystems\CTools
endif

INCLUDE_PATH = $(CTOOLS_PATH)\SCADAPack2\ISaGRAF
```

The next section sets the default compiler flags. You can add to or modify these flags. Change the default options with care, as most are required for correct operation. The flags are described in the gnu C++ compiler manual.

```
# ------------------------------------------------------------------------
# compiler flags
# ------------------------------------------------------------------------

CFLAGS = -O3 -mapcs-32 -mlittle-endian -march=armv4 -ansi -fno-builtin -DARMEL -
I"$(INCLUDE_PATH)" -DCPU=ARMARCH4 -DTOOL_FAMILY=gnu -DTOOL=gnu -std=c99
```

The next section lists the suffixes used in this make file. Generally you will not have to modify this section. Consult the gnu make documentation if you add files with new suffixes to your application.

```
# ------------------------------------------------------------------------
# list of file suffixes used in this makefile
# ------------------------------------------------------------------------

.SUFFIXES:
.SUFFIXES: .cpp .c .o .out
```

The next section describes how to make the .out file which is loaded into the SCADAPack^2 controller. Generally no changes will ever be required in this section. All the compiler options affecting this that should be changed are defined in the CFLAGS setting above.

```
# ------------------------------------------------------------------------
# rules for making .out file
# ------------------------------------------------------------------------

$(APPLICATION_NAME).out : $(objects)
# Merge all object files into one
        ccarm -I. -r -nostdlib -Wl,-X -Wl,-EL -Wl $(objects) -o tempImage.o

# Process CPP constructors and destructors
        nmarm tempImage.o | "$(CTOOLS_PATH)\Arm7\tcl\bin\tclsh84.exe"
"$(CTOOLS_PATH)\Arm7\host\x86-win32\bin\munch.tcl" -c arm > ctdt.c
        ccarm $(CFLAGS) -c -fdollars-in-identifiers ctdt.c -o ctdt.o

# Link downloadable application.
        ccarm -I. -r -nostdlib -Wl,-X -Wl,-EL -T
"$(CTOOLS_PATH)\Arm7\target\h\tool\gnu\ldscripts\link.OUT" tempImage.o ctdt.o -o
$(APPLICATION_NAME).out

# Link with CTools library to check for unresolved externals.
        ldarm -e0 tempImage.o "$(INCLUDE_PATH)\ISaGRAF_Firmware_Image" -o
tempLink.out

# Clean up temporary files
        del ctdt.c ctdt.o tempLink.out tempImage.o
```

The next section lists the dependencies of the object files on header and source files. Add additional header files and source files here. Do not add the ctools.h file to the list of dependencies.

```
# ------------------------------------------------------------------------
# list all source file dependencies here
# ------------------------------------------------------------------------

appstart.o: appstart.cpp nvMemory.h
main.o:     main.cpp nvMemory.h
```

The next section contains the rules for compiling files. Generally no changes will ever be required in this section. All the compiler options affecting this that should be changed are defined in the CFLAGS setting above

```
# ------------------------------------------------------------------------
# rules for making files
# ------------------------------------------------------------------------

%.o : %.c
        ccarm $(CFLAGS) -c $< -o $@

%.o : %.cpp
        ccarm $(CFLAGS) -c $< -o $@
```

The next section contains the rules for cleaning out all output files from a folder. Use make clean to start over from a clean slate and compile all files again. If you add additional types of output files, you will need to modify this section.

```
# ------------------------------------------------------------------------
# clean up all output files
# ------------------------------------------------------------------------
.PHONY: clean
clean:
	del *.o
	del *.out
```

Build the Application

The gnu C++ compiler is a command line compiler. To build the application:

- Open a command prompt from a shortcut or use this procedure:
  - Click **Start > Run**.
  - In *Open* type `cmd` and click **OK**.
- Switch to the folder containing the project.
  - For example type `cd c:\projects\sp2\hello`
- Type `make` and press Enter

  Make will compile the two `cpp` files, then link them into a single output file named `myApp.out`. If errors occur, they will be displayed on the command line.

Loading and Executing the C++ Application Using TelePACE

The TelePACE C\C++ Program Loader transfers executable files from a PC to the controller and controls execution of programs in the controller.

Controller Initialization

The SCADAPack2 controller should be initialized when beginning a new programming project or when it is desired to start from default conditions. It is not necessary to initialize the controller before every program load.

To completely initialize the controller, perform a Cold Boot.

When the SCADAPack2 controller starts in the cold boot mode:

- The default serial communication parameters are used.
- The TelePACE Ladder Logic application program is erased.
- The C/C++ program is erased.
- The controller is unlocked.

To perform a Cold Boot use the following procedure:

- Remove power from the SCADAPack2 controller.
- Hold down the LED POWER button.
- Apply power to the controller.
- Continue holding the LED POWER button for 25 seconds until the STAT LED begins to flash on and off continuously.

- Release the LED POWER button.

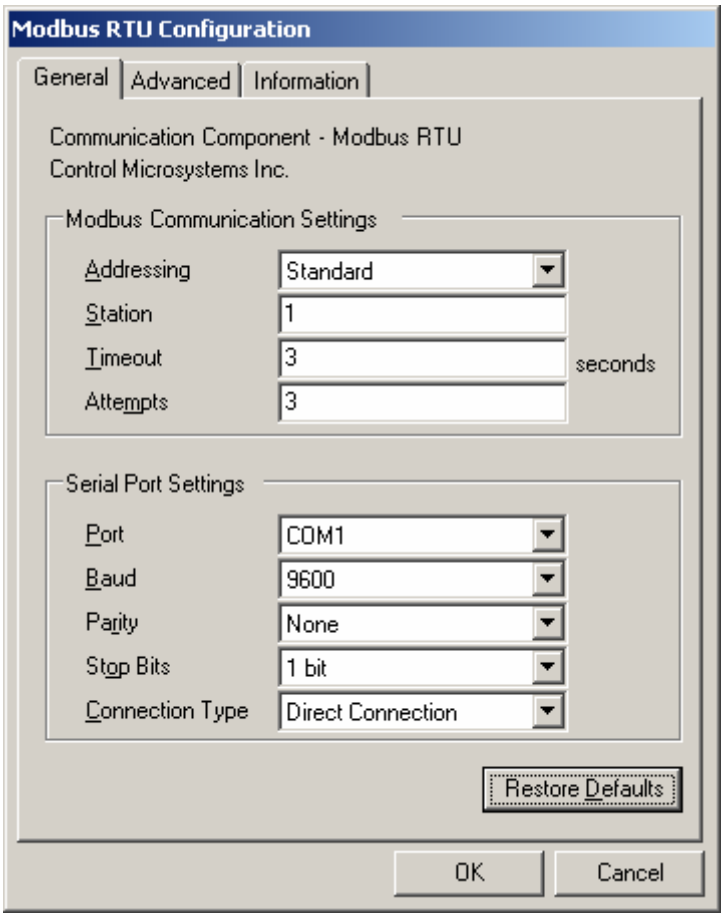
If the LED POWER button is released before the STAT LED begins to flash the SCADAPack2 controller will start in service mode, not the cold boot mode.

Connect to Controller

To connect to a controller using TelePACE firmware:

- Connect the cable to a serial port on the PC.
- Connect the cable to the com3 serial port on the controller.
- Open the TelePACE program.

To configure the PC serial port select **PC Communication Settings** from the TelePACE **Communications** menu. The PC Communications Settings dialog will appear. The default settings shown in this dialog are the same as the default serial port settings for the controller.



Use the drop down selector for the Port box to select the PC serial port being used.

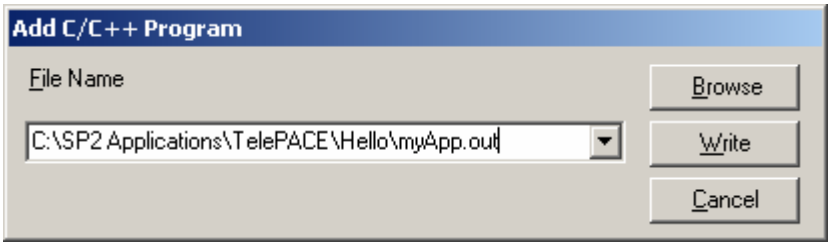
Once the desired serial communication parameters have been set click on the **OK** button.

The SCADAPack2 serial ports are set to their default parameters when a Cold Boot is done. These settings are 9600-baud, 8 data bits, no parity, 1 stop bit, Modbus RTU protocol, and station address 1.

Loading the Application

To load the `Hello` C++ application into the controller:

- From the **Controller** menu, select the **C/C++ Program Loader** command.
- Select the **Add** button and use the Browse button to locate the application. It is found at: **C:\SP2 Applications\TelePACE\Hello\myApp.out.**



- Select the **Write** button to download to the file to the controller.

Executing the Program

- Connect a terminal to *com2* on the controller. It will display the output of the program. Set the communication parameters to 9600 baud, 8 data bits, 1 stop bit, and no parity.
- From the **C/C++ Program Loader** dialog, click on the **Run** button to execute the program.

  The “hello, world” message will be displayed on the terminal.
- When multiple C++ Applications are loaded to the SCADAPack2 and the controller is power cycled, the C++ Applications are restarted in the order they were first loaded to the controller.

Loading and Executing the C++ Application Using ISaGRAF

The ISaGRAF C\C++ Program Loader transfers executable files from a PC to the controller and controls execution of programs in the controller.

Controller Initialization

The SCADAPack2 controller should be initialized when beginning a new programming project or when it is desired to start from default conditions. It is not necessary to initialize the controller before every program load.

To completely initialize the controller, perform a Cold Boot.

When the SCADAPack2 controller starts in the cold boot mode:

- The default serial communication parameters are used.
- The ISaGRAF application program is erased.
- The C program is erased.
- The controller is unlocked.

To perform a Cold Boot use the following procedure:

- Remove power from the SCADAPack2 controller.
- Hold down the LED POWER button.

- Apply power to the controller.
- Continue holding the LED POWER button for 25 seconds until the STAT LED begins to flash on and off continuously.
- Release the LED POWER button.

If the LED POWER button is released before the STAT LED begins to flash the SCADAPack2 controller will start in service mode, not the cold boot mode.

Connect to Controller

Before the project can be loaded to the SCADAPack2 controller a connection, or link, must be made between the PC and the SCADAPack2 controller.

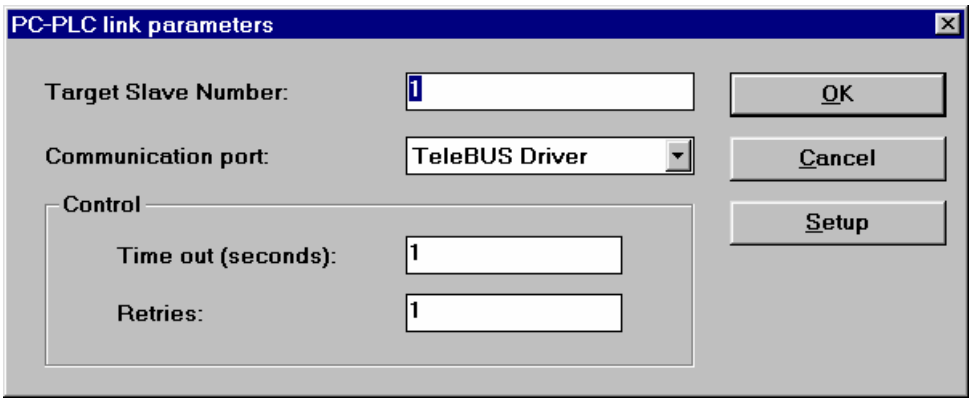
The SCADAPack2 serial ports are set to their default parameters when a Cold Boot is done. These settings are 9600-baud, 8 data bits, no parity, 1 stop bit, Modbus RTU protocol, and station address 1.

The ISaGRAF PC-PLC Link parameters define how the communication link between the PC and the target controller functions. These parameters are set to match the SCADAPack2 serial port parameters.

To open the PC_PLC link parameters dialog:

- Select **Link Setup** from the **Debug** menu.

When selected the PC-PLC Link Parameters dialog is displayed.



The **Target Slave Number:** entry is ignored when the TeleBUS Driver is selected. The TeleBUS Driver sets the target slave number. Ignore the value in this field.

- From the **Communication port:** dropdown list-box select **TeleBUS Driver**.

NOTE: If the **TeleBUS Driver** is not selectable from the **Communication port:** drop down menu then the Control Microsystems Extensions have not been installed. Refer to the installation CD jacket for installation information.

The **Time out (seconds):** edit-box sets the length of time, in seconds, to wait for a response to a command. It is an integer in the range 1 to 255 seconds. The default value is 3.

The **Retries:** edit-box sets the number of communication attempts before a message is aborted. It is an integer in the range 1 to 20. The default value is 3.

- Select the **Setup** button.

When selected the **PC Communication Settings** dialog is displayed.

- Click the Default button. This will ensure the serial parameters for the PC are the same as the parameters on each of the SCADAPack2 serial ports.
- In the Port dropdown selection select the serial port you are using on your PC to communicate with the SCADAPack2 controller.
- Connect SCADAPack2 com3 to the PC serial port using an RS-232 serial communication cable. This cable is a null modem or computer-to-computer cable.

Loading the Application

To load the `Hello` C++ application into the controller:

- From the **Controller** menu, select the **C/C++ Program Loader** command.
- Select the **Add** button and use the Browse button to locate the application. It is found at: **C:\SP2 Applications\ISaGRAF\Hello\myApp.out.**

- Select the **Write** button to download to the file to the controller.

Executing the Program

- Connect a terminal to *com2* on the controller. It will display the output of the program. Set the communication parameters to 9600 baud, 8 data bits, 1 stop bit, and no parity.
- From the **C/C++ Program Loader** dialog, click on the **Run** button to execute the program.

  The “hello, world” message will be displayed on the terminal.
- When multiple C++ Applications are loaded to the SCADAPack2 and the controller is power cycled, the C++ Applications are restarted in the order they were first loaded to the controller.

C++ Program Development

Program Architecture

This section of the manual describes the process for developing end-user applications in C++ for the SCADAPack2 controller family. The SCADAPack2 C++ Tools are based on the ***GNU Compiler Collection (GCC)*** for the Arm7 processor. Users will be able to create, compile and debug applications using these tools.

Application Startup

There are two files associated with the startup structure: `appstart.cpp` and `nvMemory.h`. Each is described below.

Application Startup Function (appstart.app)

The start-up code has the following major functions:

- initialize application program variables;
- execute the `main()` function

Source code for the `appstart` function is supplied with the C++ Tools sample application in the file `appstart.cpp`. The following discussion refers to statements found in this file. At the top of `appstart.cpp` are initialized global variables used to configure settings for the main task. The default values are suitable for most applications.

```
/* ------------------------------------------------------------------
   Global Variables
   ------------------------------------------------------------------ */
// These parameters are used when the task main() is created.

// Priority of the task main().
// Priority 100 is recommended for a continuously running task.
// A task with priority > 100 will never be given the CPU.
// See manual for details.
UINT32 mainPriority = 100;

// Stack space allocated to the task main().
// Note that at least 5 stack blocks are needed when calling fprintf().
UINT32 mainStack = 5;

// Application group assigned to the task main().
// A unique value is assigned by the system to the applicationGroup
// for this application. Use this variable in all calls to create_task()
// by this application. See manual for details.
UINT32 applicationGroup = 0;

// Pointer to static non-volatile data.
// Define the structure NV_MEMORY in nvMemory.h
NV_MEMORY * pNvMemory = NULL;

// Size of structure in static non-volatile memory
UINT32 nvMemorySize = sizeof(NV_MEMORY);

// applicationType and applicationTypeLimit may be used to limit
// the number of executable instances of this application.
// Valid values for applicationType are 0 to 65535. Default type is 0.
// Valid values for applicationTypeLimit are 0 to 32.
// Default limit is 0 which = no limit
```

```
UINT32 applicationType      = 0;      // valid types : 0 to 65535
UCHAR  applicationTypeLimit = 0;      // valid limits: 0 to 32; 0 = no limit
```

mainPRiority

The variable mainPriority selects the priority for the task main. The task main is declared in the file main.cpp. There are 255 priority levels, and the highest priority task has a priority of 0. The table below lists the recommended priority values to use with the SCADAPack2. Note that the logic application executes in a continuous loop at priority 100. This means that a task selected with priority > 100 will never be given the CPU. Priority 100 is suitable for most C++ Applications.

Priority Description	Priority Value for SCADAPack2	Recommended Use
Higher Priority	25	Not recommended
	50	Serial protocol tasks
	75	IP protocol tasks or other blocking task (e.g. wait_event called each loop)
Lower Priority	100	Any continuously running loop (e.g. I/O processing)

mainStack

The variable mainStack selects the stack space for the task main. Note that at least 5 stack blocks are needed when the main task calls the function fprintf. The heap size is not configurable. The C++ application has access to the entire system heap.

applicationGroup

The variable applicationGroup is assigned with a unique value by the operating system to identify each user-defined C++ application. The variable applicationGroup should be used for the parameter type when calling the function create_task. When an application is stopped or deleted, all tasks created by the same application group will be stopped.

pNvMemory and nvMemorySize

The variables pNvMemory and nvMemorySize are declared next and do not require changes. The structure NV_MEMORY is defined in the file nvMemory.h and is discussed in the next section.

applicationType and applicationTypeLimit

The variables applicationType and applicationTypeLimit may be used to limit the number of instances of a C++ Application that may be executed on the same SCADAPack2. For example, to load another instance of a C++ Application, simply rename the application file before loading it to the SCADAPack2. By default, there is no instance limit set. To limit the number of instances to one, for example, select a unique value for applicationType and set applicationTypeLimit = 1.

appstart

The appstart function is the entry point for the C++ Application. This function begins by initializing the global pointer to static non-volatile data. The main task is called next. If the main task returns, the application including all tasks created by main is ended.

Non-Volatile Memory (nvMemory.h)

SCADAPack2 C++ Applications may declare variables as non-volatile by locating them in SRAM. There is 8 KB of SRAM available for static non-volatile variables. And if this is not enough, up to 1 MB of SRAM is available for dynamic non-volatile memory allocation. For more details see the function `allocateMemory`.

Only non-initialized variables are defined as non-volatile. Initialized variables do not need to be non-volatile, since they are initialized to the same value on application startup.

The following example describes the procedure for declaring non-volatile variables. Consider the following C++ Application defined in the two files: `main.cpp` and `file2.cpp`.

Version 1

The first version of these files defines which non-volatile variables are required for each file. Note that local and module variables would normally exist as well.

main.cpp:

```
#include "ctools.h"

// Non-volatile variables required by main.cpp
static UINT32 variable1;
static UCHAR array1[20];
static struct sample table[10];

void main(void)
{
        variable1 = array1[0] * table[0].index;
}
```

file2.cpp:

```
#include "ctools.h"

// Non-volatile variables required by file2.cpp
static UINT32 variable2;

void function1(void)
{
        variable2++;
}
```

Version 2

This second version of these files shows how to declare these variables as non-volatile. To do this the declarations have been moved to the header file `nvMemory.h` and are shown in bold below. A template for `nvMemory.h` is provided in the sample C++ Application. This header file must be included in each file that accesses the non-volatile variables.

The only undesirable effect of making certain variables non-volatile is that these variables must become global variables. To access the non-volatile variables in code use the pointer, `pNvMemory`, to the NV_MEMORY structure as shown below.

main.cpp:

```
#include "ctools.h"
#include "nvMemory.h"

void main(void)
```

```
{
        pNvMemory–>variable1 =  pNvMemory–>array1[0] *
                                pNvMemory–>table[0].index;
}
```

file2.cpp:

```
#include "ctools.h"
#include "nvMemory.h"

void function1(void)
{
        pNvMemory–>variable2++;
}
```

nvMemory.h:

```
/* ------------------------------------------------------------------
   nvMemory.h
   Global definitions for user variables that need to be non-volatile.
   Copyright 2006, Control Microsystems Inc.
   ------------------------------------------------------------------ */

/* Prevent multiple inclusions */
#ifndef NVMEMORY_H
#define NVMEMORY_H

#ifdef __cplusplus
extern "C"
{
#endif

// ------------------------------------------------------
// Include-files
// ------------------------------------------------------
#include "ctools.h"

/* ------------------------------------------------------------------
   Variables located in Static Non-Volatile Memory
   ------------------------------------------------------------------ */
// Add fields to this global structure for variables used in your
// application file(s) that need to be non-volatile. Include
// nvMemory.h in all files that use the variable pNvMemory to access
// NV memory.

typedef struct s_nvMemory
{
        UCHAR dummyVariable;

        // Add fields here for variables used in your application
        // file(s) that need to be non-volatile.

        // Non-volatile variables required by main.cpp
        UINT32 variable1;
        UCHAR array1[20];
        struct sample table[10];

        // Non-volatile variables required by file2.cpp
        float variable2;

}NV_MEMORY;

// Pointer to static non-volatile data
extern NV_MEMORY * pNvMemory;

#ifdef __cplusplus
}
#endif
```

```
#endif // NVMEMORY_H
```

GNU Compiler Options

The GNU C++ compiler is installed with the SCADAPack2 C++ Tools. The `build.bat` file included in the sample C++ application uses the following command line for compiling:

```
ccarm -O3 -mapcs-32 -mlittle-endian -march=armv4 -ansi
-fno-builtin -DARMEL -I"%CTOOLS_PATH%" -DCPU=ARMARCH4
-DTOOL_FAMILY=gnu -DTOOL=gnu -std=c99 -c main.cpp
```

These compiler options are described in the table below. The complete list of compiler options is may be found in the document *Using the GNU Compiler Collection (GCC)* which is installed with the compiler at C:\Program Files\Control Microsystems\CTools\Arm7\gcc.pdf.

Option	Description
`-O3`	Level 3 optimization
`-mapcs-32`	Generate code for a processor running with a 32-bit program counter, and conforming to the function calling standards for the APCS 32-bit option.
`-mlittle-endian`	Generate code for a processor running in little-endian mode.
`-march=armv4`	Specifies the name of the target ARM architecture as armv4.
`-ansi -std=c99`	ISO C99 language standard for C++
`-fno-builtin`	Don't recognize built-in functions that do not begin with '__builtin_' as prefix.
`-D`*`name`*	Predefine *name* as a macro with the definition 1.
`-D`*`name=definition`*	Predefine *name* as a macro with *definition.*
`-c`	Compile or assemble the source files, but do not link.
`-I`*`dir`*	Add the directory *dir* to the head of the list of directories to be searched for header files. If you use more than one '-I' option, the directories are scanned in left-to-right order; the standard system directories come after.
`-fdollars-in-identifiers`	Accept '$' in identifiers.
`-o`*`file`*	`Specifies the name of the output file.`

Application Development

Please refer to the ***Program Development Tutorial*** for details on how to build, load and execute a C++ Application.

Real Time Operating System

The real time operating system (RTOS) provides the programmer with tools for building sophisticated applications. The RTOS allows pre-emptive scheduling of event driven tasks to provide quick response to real-world events. Tasks multi-task cooperatively. Inter-task communication and event notification functions pass information between tasks. Resource functions facilitate management of non-sharable resources.

Task Management

The task management functions provide for the creation and termination of tasks. Tasks are independently executing routines. The RTOS uses a cooperative multi-tasking scheme, with pre-emptive scheduling of event driven tasks.

The initial task (the **main** function) may create additional tasks. The maximum number of tasks is limited only by available memory. There are 256 task priority levels to aid in scheduling of task execution.

Task Execution

SCADAPack2 controllers can execute one task at a time. The RTOS switches between the tasks to provide parallel execution of multiple tasks. The application program can be event driven, or tasks can execute round-robin (one after another).

Task execution is based upon the priority of tasks. There are 256 priority levels. Application programs can use levels 100 to 0. The main task is created at priority level 100. Task level 0 is the highest priority task.

Tasks that are not running are held in queues. The Ready Queue holds all tasks that are ready to run. Event queues hold tasks that are waiting for events. Message queues hold tasks waiting for messages. Resource queues hold tasks that are waiting for resources. The envelope queue holds tasks that are waiting for envelopes.

Priority Inversion Prevention

When a higher priority task, Task H, requests a resource, which is already obtained by a lower priority task, Task L, the higher priority task, is blocked until Task L releases the resource. If Task L is unable to execute to the point where its releases the resource, Task H will remain blocked. This is called a Priority Inversion.

To prevent this from occurring, the prevention method known as Priority Inheritance has been implemented. In the example already described, the lower priority task, Task L, is promoted to the priority of Task H until it releases the needed resource. At this point Task L is returned to its original priority. Task H will obtain the resource now that it is available.

Note that this does not prevent deadlocks that occur when each task requests a resource that the other has already obtained. This “deadly embrace” is a design error in the application program.

Operating System Scheduling

The SCADAPack2 operating system supports a round-robin scheduling algorithm combined with pre-emptive priority scheduling. It shares the CPU fairly among all ready tasks of the same priority. Round-robin scheduling uses time slicing to achieve fair

allocation of the CPU to all tasks with the same priority. Each task, in a group of tasks with the same priority, executes for a defined interval or time slice.

Because the time slicing is performed by the kernel of the operating system, it is not necessary anymore for the tasks to call explicitely **release_processor** to release CPU time to other tasks of the same priority. In contrary it can harm. When a task expects a fair share of the CPU, calling release_processor before the end of the time slice puts it immediately at the end of round-robin-queue. Therefore the CPU time share can be significantly reduced. The function release_processor still makes sense if the calling task does not have anything to do for the moment.

A new function **sleep_processor** is introduced to release CPU for a certain time.

Task Management Functions

There are five RTOS functions for task management. Refer to the ***Function Specification*** section for details on each function listed.

create_task	Create a task and make it ready to execute.
end_task	Terminate a task and free the resources and envelopes allocated to it.
end_application	Terminate all application program type tasks. This function is used by communication protocols to stop the application program prior to loading new code.
installExitHandler	Specify a function that is called when a task is ended with the end_task or end_application functions.
getTaskInfo	Return information about a task.

Task Management Structures

The **ctools.h** file defines the structure **Task Information Structure** for task management information. Refer to the ***C Tools Structures and Types*** section for complete information on structures and enumeration types.

Resource Management

The resource management functions arbitrate access to non-sharable resources. These resources include physical devices such as serial ports, and software that is not re-entrant.

The RTOS defines nine system resources, which are used by components of the I/O drivers, memory allocation functions and communication protocols.

An application program may define other resources as required. Care must be taken not to duplicate any of the resource numbers declared in **ctools.h** as system resources.

Resource Management Functions

There are three RTOS functions for resource management. Refer to the ***Function Specification*** section for details on each function listed.

request_resource	Request access to a resource and wait if the resource is not available.
poll_resource	Request access to a resource. Continue execution if the resource is not available

release_resource Free a resource for use by other tasks.

IO_SYSTEM Resource

The IO_SYSTEM resource regulates access to all functions using the I/O system. C application programs, ladder logic programs, communication protocols and background I/O operations share the I/O system. It is imperative the resource is obtained to prevent a conflict, as protocols and background operations are interrupt driven. Do not retain control of the resource for more that 0.1 seconds, or background operations will not execute properly.

DYNAMIC_MEMORY Resource

The DYNAMIC_MEMORY resource regulates access to all memory allocation functions. These functions allocate memory from the system heap. The heap is shared amongst all tasks. The allocation functions are non-reentrant.

The DYNAMIC_MEMORY resource must be obtained before using any of the following functions.

calloc allocates data space dynamically

free frees dynamically allocated memory

malloc allocates data space dynamically

realloc changes the size of dynamically allocated space

Inter-task Communication

The inter-task communication functions pass information between tasks. These functions can be used for data exchange and task synchronization. Messages are queued by the RTOS until the receiving task is ready to process the data.

Inter-task Communication Functions

There are five RTOS functions for inter-task communication. Refer to the ***Function Specification*** section for details on each function listed.

send_message Send a message envelope to another task.

receive_message Read a received message from the task's message queue or wait if the queue is empty.

poll_message Read a received message from the task's message queue. Continue execution of the task if the queue is empty.

allocate_envelope Obtain a message envelope from free pool maintained by the RTOS, or wait if none is available.

deallocate_envelope Return a message envelope to the free pool maintained by the RTOS.

Inter-task Communication Structures

The **ctools.h** file defines the structure **Message Envelope Structure** for inter-task communication information. Refer to the ***C Tools Structures and Types*** section for complete information on structures and enumeration types.

Event Notification

The event notification functions provide a mechanism for communicating the occurrence of events without specifying the task that will act upon the event. This is different from inter-task communication, which communicates to a specific task.

Multiple occurrences of a single type of event are queued by the RTOS until a task waits for or polls the event.

Event Notification Functions

There are four RTOS functions for event notification. Refer to the ***Function Specification*** section for details on each function listed.

wait_event Wait for an event to occur.

poll_event Check if an event has occurred. Continue execution if one has not occurred.

signal_event Signal that an event has occurred.

interrupt_signal_event Signal that an event has occurred from an interrupt handler. This function must only be called from within an interrupt handler.

There are two support functions, which are not part of the RTOS that may be used with events.

startTimedEvent Enables signaling of an event at regular intervals.

endTimedEvent Terminates signaling of a regular event.

System Events

The RTOS defines events for communication port management and background I/O operations. An application program may define other events as required. Care must be taken not to duplicate any of the event numbers declared in **ctools.h** as system events.

BACKGROUND This event triggers execution of the background I/O routines. An application program cannot use it.

COM1_RCVR This event is used by communication protocols to signal a character or message received on com1. It can be used in a custom character handler (see **install_handler**).

COM2_RCVR This event is used by communication protocols to signal a character or message received on com2. It can be used in a custom character handler (see **install_handler**).

COM3_RCVR This event is used by communication protocols to signal a character or message received on com3. It can be used in a custom character handler (see **install_handler**).

COM4_RCVR This event is used by communication protocols to signal a character or message received on com4. It can be used in a custom character handler (see **install_handler**).

NEVER This event is guaranteed never to occur. It can be used to disable a task by waiting for it to occur. However, to end a task it is better to use **end_task**. This frees all resources and stack space allocated to the task.

Error Reporting

Sharable I/O drivers to return error information to the calling task use the error reporting functions. These functions ensure that an error code generated by one task is not reported in another task. The **errno** global variable used by some functions may be modified by another task, before the current task can read it.

Error Reporting Functions

There are two RTOS functions for error reporting. Refer to the ***Function Specification*** section for details on each function listed.

check_error Check the error code for the current task.

report_error Set the error code for the current task.

RTOS Example Application Program

The following program is used in the explanation of the RTOS functions. It creates several simple tasks that demonstrate how tasks execute. A task is a C language function that has as its body an infinite loop so it continues to execute forever.

The `main` task creates two tasks. The `echoData` task is higher priority than `main`. The `auxiliary` task is the same priority as `main`. The `main` task then executes round robin with other tasks of the same priority.

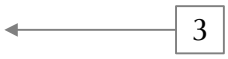
The `auxiliary` task is a simple task that executes round robin with the other tasks of its priority. Only the code necessary for task switching is shown to simplify the example.

The `echoData` task waits for a character to be received on a serial port, then echoes it back out the port. It waits for the event of the character being received to allow lower priority tasks to execute. It installs a character handler function – `signalCharacter` – that signals an event each time a character is received. This function is hooked into the receiver interrupt handler for the serial port.

The execution of this program is explained in the ***Explanation of Task Execution*** section.

```
/* ------------------------------------------------------------------
   SCADAPack2 Real Time Operating System Sample
   Copyright (c) 2006, Control Microsystems Inc.

   This program creates several simple tasks for demonstration of the
   functionality of the real time operation system.
   ------------------------------------------------------------------ */

#include <stdio.h>
#include <ctools.h>

/* ------------------------------------------------------------------
   Constants
   ------------------------------------------------------------------ */

#define CHARACTER_RECEIVED    10

/* ------------------------------------------------------------------
   signalCharacter

   The signalCharacter function signals an event when a character is
   received. This function must be called from an interrupt handler.
   ------------------------------------------------------------------ */

void signalCharacter(UINT16 character, UINT16 error)
{
```

```
        /* If there was no error, signal that a character was received */
        if (error == 0)
        {
                interrupt_signal_event(CHARACTER_RECEIVED);
        }

        /* Prevent compiler unused variables warning (generates no code) */
        character;
}

/* ---------------------------------------------------------------------
   echoData

   The echoData function is a task that waits for a character
   to be received on com1 and echoes the character back. It installs
   a character handler for com1 to generate events on the reception
   of characters.
   --------------------------------------------------------------------- */

void echoData(void)
{
        struct prot_settings protocolSettings;
        struct pconfig portSettings;
        int character;

        /* Disable communication protocol */
        get_protocol(com1, &protocolSettings);
        protocolSettings.type = NO_PROTOCOL;
        set_protocol(com1, &protocolSettings);

        /* Set serial communication parameters */
        portSettings.baud      = BAUD9600;
        portSettings.duplex    = FULL;
        portSettings.parity    = NONE;
        portSettings.data_bits = DATA8;
        portSettings.stop_bits = STOP1;
        portSettings.flow_rx   = RFC_MODBUS_RTU;
        portSettings.flow_tx   = TFC_NONE;
        portSettings.type      = RS232;
        portSettings.timeout   = 600;
        set_port(com1, &portSettings);

        /* Install handler for received character */
        install_handler(com1, signalCharacter);

        while (TRUE)
        {
                /* Wait for a character to be received */
                wait_event(CHARACTER_RECEIVED);

                /* Echo the character back */
                character = fgetc(com1);
                if (character == EOF)
                {
                        // clear overflow error flag to re-enable com1
                        clearerr(com1);
                }
                fputc(character, com1);
        }
}


/* ---------------------------------------------------------------------
   auxiliary

   The auxiliary function is a task that performs some action
   required by the program. It does not have specific function so
   that the real time operating system features are clearer.
   --------------------------------------------------------------------- */

void auxiliary(void)
```



```
{
        while (TRUE)                                                    7
        {
                /* ... add application specific code here ... */

                /* Allow other tasks of this priority to run */
                release_processor();
        }
}

/* ---------------------------------------------------------------------
   main

   This function creates two tasks: one at priority three and one at
   priority 1 to demonstrate the functions of the RTOS.
   --------------------------------------------------------------------- */

                                                                        1
void main(void)
{                                                                       2
        /* Create serial communication task */
        create_task(echoData, 3, APPLICATION,
3);

        /* Create a task - same priority as main() task */
        create_task(auxiliary, 1, APPLICATION, 2);

                                                                        5
        while (TRUE)
        {
                /* ... add application specific code here ... */

                /* Allow other tasks of this priority to execute */
                release_processor();                                    6
        }
}
```

Explanation of Task Execution

SCADAPack2 controllers can execute one task at a time. The Real Time Operating System (RTOS) switches between the tasks to provide parallel execution of multiple tasks. The application program can be event driven, or tasks can execute round-robin (one after another). This program illustrates both types of execution.

Task execution is based upon the priority of tasks. There are 256 priority levels. Level 255 is reserved for the `null` task. This task runs when there are no other tasks available for execution. Application programs can use levels 100 to 0. The main task is created at priority level 100.

Tasks that are not running are held in queues. The Ready Queue holds all tasks that are ready to run. Event queues hold tasks that are waiting for events. Message queues hold tasks waiting for messages. Resource queues hold tasks that are waiting for resources. The envelope queue holds tasks that are waiting for envelopes.

The execution of the tasks is illustrated by examining the state of the queues at various points in the program. These points are indicated on the program listing above. The examples show only the Ready queue, the Event 10 queue and the executing task. These are the only queues relevant to the example.

Execution Point 1

This point occurs just before the `main` task begins. The `main` task has not been created by the RTOS. The `null` task has been created, but is not running. No task is executing.

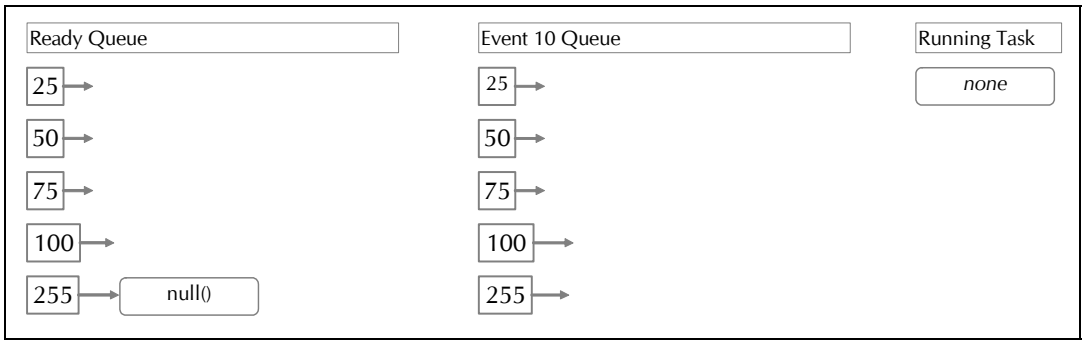


Figure 1: Queue Status before Execution of main Task

Execution Point 2

TODO: Update the remaining queue diagrams below for priority levels 255 to 25.

This point occurs just after the creation of the `main` task. It is the running task. On the next instruction it will create the `echoData` task.

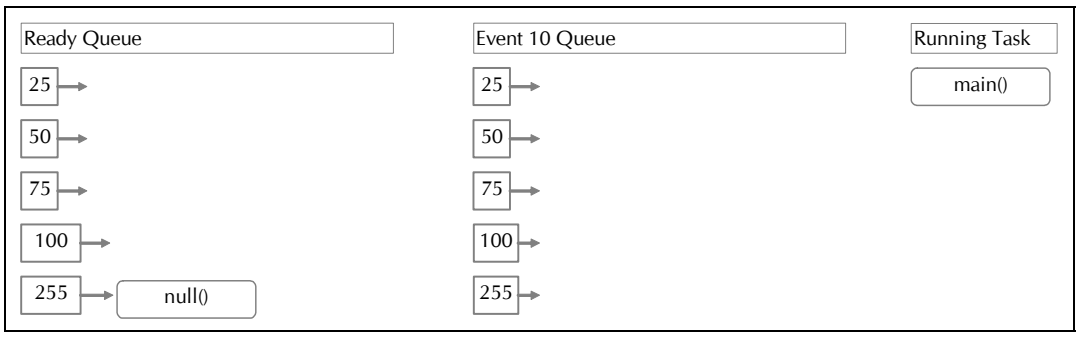


Figure 2: Queue Status at Start of `main` Task

Execution Point 3

This point occurs just after the `echoData` task is created. The `echoData` task is higher priority than the `main` task so it is made the running task. The `main` task is placed into the ready queue. It will execute when it becomes the highest priority task.

The `echoData` task initializes the serial port and installs the serial port handler function `signalCharacter`. It will then wait for an event. This will suspend the task until the event occurs.

The `signalCharacter` function will generate an event each time a character is received without an error.

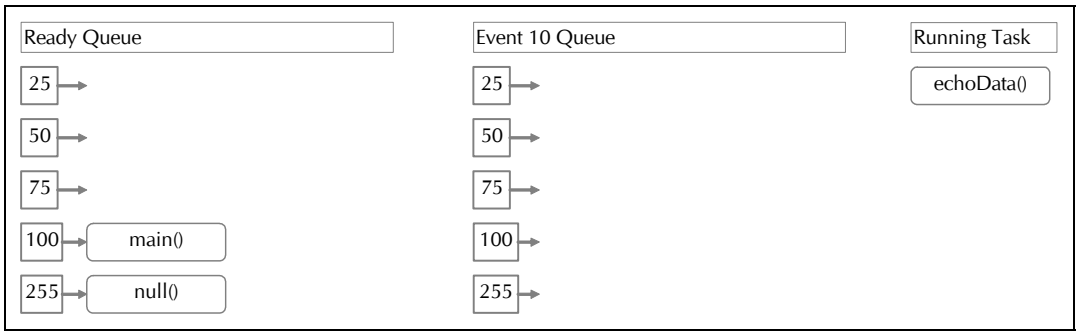


Figure 3: Queue Status after Creation of* `echoData` *Task

Execution Point 4

This point occurs just after the `echoData` task waits for event 10. It has been placed on the event queue for event 10.

The highest priority task on the ready queue was the `main` task. It is now running. On the next instruction it will create another task at the same priority as `main`.

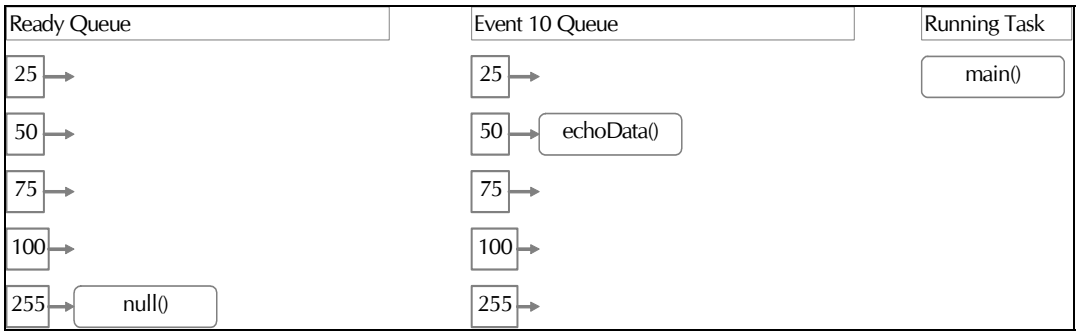


Figure 4: Queue Status After* `echoData` *Task Waits for Event

Execution Point 5

This point occurs just after the creation of the **auxiliary** task. This task is the same priority as the **main** task. Therefore the **main** task remains the running task. The **auxiliary** task is ready to run and it is placed on the Ready queue.

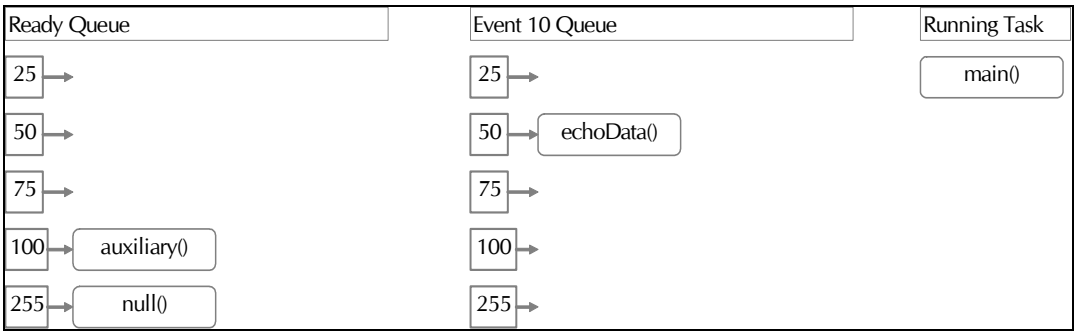


Figure 5 Queue Status after Creation of* `auxiliary` *Task

Execution Point 6

This point occurs just after the `main` task releases the processor, but before the next task is selected to run. The `main` task is added to the end of the priority 1 list in the Ready queue.

On the next instruction the RTOS will select the highest priority task in the Ready queue.

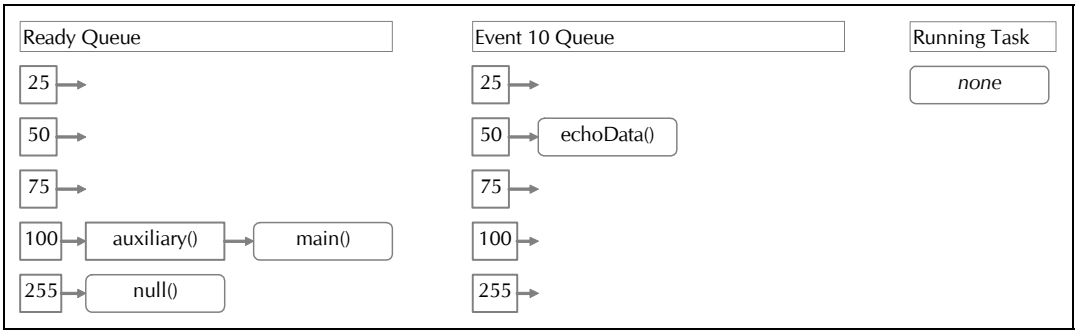


Figure 6: Queue Status After* `main` *Task Releases Processor

Execution Point 7

This point is just after the `auxiliary` task has started to run. The `main` and `auxiliary` tasks will continue to alternate execution, as each task releases the processor to the other.

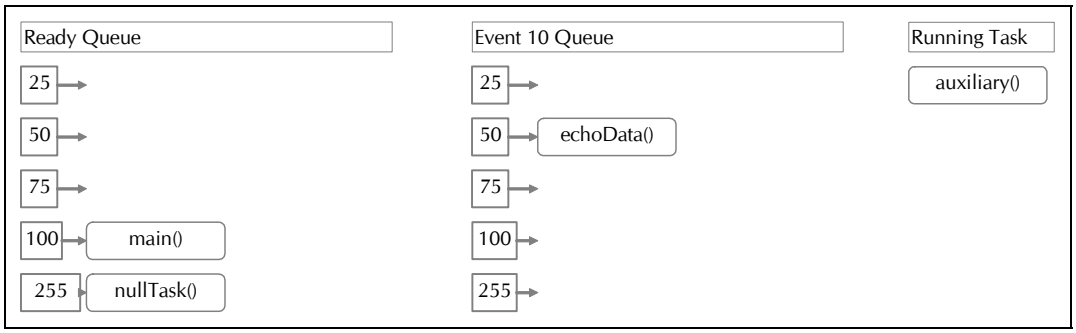


Figure 7: Queue Status at Start of `auxiliary` Task

Execution Point 8

This point occurs just after a character has been received. The `signalCharacter` function executes and signals an event. The RTOS checks the event queue for the event, and makes the highest priority task ready to execute. In this case the `echoData` task is made ready.

The RTOS then determines if the new task is higher priority than the executing task. Since the `echoData` task is higher priority than the `auxiliary` task, a task switch occurs. The `auxiliary` task is placed on the Ready queue. The `echoData` task executes.

Note the position of `auxiliary` in the Ready queue. The `main` task will execute before it at the next task switch.

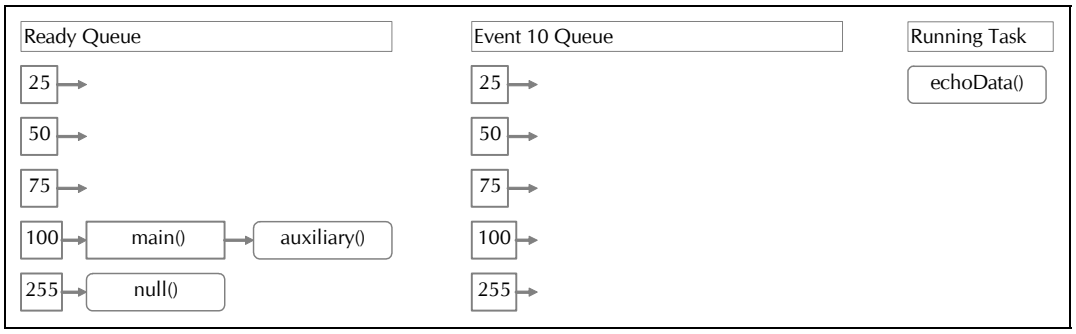


Figure 8: Queue Status after Character Received

Execution Point 9

This point occurs just after the `echoData` task waits for the character-received event. It is placed on the event 10 queue. The highest priority task on the ready queue – `main` – is given the processor and executes.

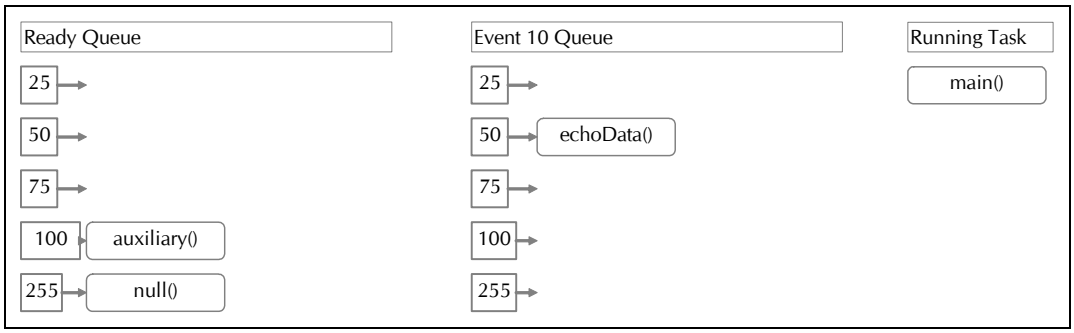


Figure 9: Queue Status after* `echoData` *Waits for Event

Overview of Programming Functions

This section of the User Manual provides an overview of the Functions, Macros, Structure and Types available to the user. The Functions, Macros, Structure and Types overview is separated into sections of related functions. Refer to the Function Specification, C Tools Macros and C Tools Structures and Types sections of this manual for detailed explanations of the Functions, Macros, Structure and Types described here.

Controller Operation

This section of the manual provides an overview of the ISaGRAF functions relating to controller operation. These functions are provided in addition to the run-time library supplied with the Hitachi Embedded Workshop.

Start Up Functions

The following functions are called by the application startup function **appstart**. They are intended for use only in the context of **appstart.** Refer to the ***Function Specification*** section for details on each function listed.

startup_task	Returns the address of the system start up routine.
runBackgroundIO	Starts or stops the Background I/O task.
runTarget	Starts or stops the run-time engine task.
initializeApplicationVariables	Initializes user application variables.
runIOSystem	Starts or stops the I/O system.
start_protocol	Starts serial protocol according to stored parameters.
mTcpRunServer	Starts or stops the Modbus/TCP Server task.
runMasterIpStartTask	Starts or stops the Modbus/TCP Master support task.
runBackgroundIO	Starts or stops background I/O task (e.g. Dialup support, pushbutton LED power control).
runTarget	Starts or stops the run-time engine (Ladder Logic or ISaGRAF)
executeConstructors	Execute all user-created global class object constructors.
executeDestructors	Execute all user-created global class object destructors.

Start Up Macros

The **ctools.h** file defines the following macros for use with the start up task. Refer to the ***C Tools Macros*** section for details on each macro listed.

STARTUP_APPLICATION	Specifies the application start up task.
STARTUP_SYSTEM	Specifies the system start up task.

Start Up Task Info Structure

The **ctools.h** file defines the structure **TASKINFO** for use with the startup_task function. Refer to the ***C Tools Structures and Types*** section for complete information on structures and enumeration types.

Program Status Information Functions

There are two library functions related to controller program status information. Refer to the ***Function Specification*** section for details on each function listed.

getProgramStatus Returns the application program execution status.

setProgramStatus Sets the application program execution status.

Controller Information Functions

There are no functions related to controller information. Refer to the ***Function Specification*** section for details.

getControllerID Get the controller ID code.

Firmware Version Information Functions

There is one function related to the controller firmware version. Refer to the ***Function Specification*** section for details.

getVersion Returns controller firmware version information.

Firmware Version Information Structure

The **ctools.h** file defines the structure **Version Information Structure** for controller firmware version information. Refer to the ***C Tools Structures and Types*** section for complete information on structures and enumeration types.

Configuration Data Flash Memory Functions

SCADAPack2 controllers use flash memory to store controller settings. The flash memory functions have one parameter: flags indicating which areas to store into flash. A sum of more than one area may be selected. Valid flags are listed below and defined in ctools.h.

Area Flag	Loaded on Reset	Controller Settings in this Area
CS_ETHERNET	always	Ethernet MAC address
CS_OPTIONS	always	Controller factory options.
CS_PERMANENT	Saved settings loaded on Service and Run Boot. Replaced with default settings on Cold Boot.	Controller type, IP address, Gateway, Network mask, IP Configuration mode, Lock state and password, I/O System settings, I/O error indication setting TelePACE Firmware only: Register assignment, Outputs on stop settings
CS_RUN	Saved settings loaded on Run Boot. Default settings loaded on	Serial port settings, Serial protocol settings, Modbus/TCP settings, HART I/O settings, LED power

Area Flag	Loaded on Reset	Controller Settings in this Area
	Service Boot. Replaced with default settings on Cold Boot.	settings, Store and forward table

There are two library functions related to the configuration data flash memory. Refer to the ***Function Specification*** section for details on each function listed.

flashSettingsLoad This function stores the controller settings in the indicated area or areas to flash memory.

flashSettingsSave This function loads the controller settings in the indicated area or areas from flash memory.

System Functions

The **ctools.h** file defines the following functions for system initialization and for retrieving system information. Some of these functions are primarily used in the **appstart.c** routine, having limited use in an application program.

Refer to the ***Function Specification*** section for details on each function listed.

ioClear Clears all I/O points

ioDatabaseReset Resets the controller to default settings.

ioRefresh Refresh outputs with internal data

ioReset Reset all I/O modules

Controller I/O Hardware

This section of the manual provides an overview of the ISaGRAF C Tools functions relating to controller signal input and output (I/O).

Analog Input Functions

The controller supports internal analog inputs and external analog input modules. Refer to the ***SCADAPack2 System Hardware Manual*** for further information on controller analog inputs and analog input modules.

There are several library functions related to internal analog inputs and analog input modules. Refer to the ***Function Specification*** section for details on each function listed.

readBattery Read the controller RAM battery voltage.

readThermistor Read the controller ambient temperature sensor.

ioRead4Ain read 4 analog inputs into I/O database.

ioRead8Ain read 8 analog inputs into I/O database.

ioRead5505Inputs Read the digital and analog inputs from a 5505 I/O Module.

ioRead5505Outputs Read the configuration data from a 5505 I/O Module.

ioRead5506Inputs Read the digital and analog inputs from a 5506 I/O Module.

ioRead5506Outputs Read the configuration data from a 5506 I/O Module.

ioWrite5505Outputs Write the configuration data to a 5505 I/O Module.

ioWrite5506Outputs Write the configuration data to a 5506 I/O Module.

ioRead5601Inputs Read the digital and analog inputs from a SCADAPack 5601 I/O Module.

ioRead5604Inputs Read the digital and analog inputs from a SCADAPack 5604 I/O Module.

ioRead5606Inputs Read the digital and analog inputs from a 5606 I/O Module.

ioRead5606Outputs Read the digital and analog outputs from a 5606 I/O Module.

Analog Output Functions

The controller supports external analog output modules. Refer to the ***SCADAPack2 System Hardware Manual*** for further information on these modules.

There are three library functions related to analog output modules. Refer to the ***Function Specification*** section for details on each function listed.

ioRead5606Outputs Read the digital and analog outputs from a 5606 I/O Module.

ioReadAout2 Read buffered data for 2 point analog output module

ioReadAout4 Read buffered data for 4 point analog output module

ioReadAout5303 Read buffered data for 5303 analog output module

ioWriteAout2 Write buffered data for 2 point analog output module

ioWriteAout4 Write buffered data for 4 point analog output module

ioWriteAout5303 Write buffered data for 5303 analog output module

ioWrite5606Outputs Write to the digital and analog outputs of a 5606 I/O Module.

Digital Input Functions

The controller supports internal digital inputs and external digital input modules. Refer to the ***SCADAPack2 System Hardware Manual*** for further information on controller digital inputs and digital input modules.

There are several library functions related to digital inputs and external digital input modules. Refer to the ***Function Specification*** section for details on each function listed.

ioRead5606Inputs Read the digital and analog inputs from a 5606 I/O Module.

ioReadDin5232 Read buffered data from the 5232 digital inputs

ioReadCounter5232 Read buffered data from the 5232 counter inputs.

ioReadDin16 Read buffered data from any 16 point Digital input module.

ioReadDin32 Read buffered data from any 32 point Digital input module.

ioRead5601Inputs Read buffered data from the digital and analog inputs of a 5601 I/O module.

ioRead5604Inputs Read the digital and analog inputs from a SCADAPack 5604 I/O Module.

ioReadDin8 Read buffered data from any 8 point Digital input module.

Digital Output Functions

The controller supports external digital output modules. Refer to the ***SCADAPack2 System Hardware Manual*** for further information on controller digital output modules.

There are several library functions related to digital output modules. Refer to the ***Function Specification*** section for details on each function listed.

ioRead5606Inputs Read the digital and analog inputs from a 5606 I/O Module.

ioReadDout16 Read buffered data from any 16 point Digital output module.

ioReadDout32 Read buffered data from any 32 point Digital output module.

ioRead5601Outputs Read buffered data from any 5601 I/O Module.

ioRead5604Outputs Read buffered data from any 5604 I/O Module.

ioReadDout8 Read buffered data from any 8 point Digital output module.

ioWriteDout16 Write data to the I/O tables for any 16 point Digital output module.

ioWriteDout32 Write data to the I/O tables for any 32 point Digital output module.

ioWrite5601Outputs Write data to the I/O table for the digtal outputs of a 5601 I/O Module (SCADAPack2 lower I/O module).

ioWrite5604Outputs Write to the digital and analog outputs of SCADAPack 5604 I/O Module.

ioWrite5606Outputs Write to the digital and analog outputs of a 5606 I/O Module.

ioWriteDout8 Write data to the I/O tables for any 8 point Digital output module.

Counter Input Functions

The controller supports internal counters and external counter modules. The counter registers are 32 bits, for a maximum count of 4,294,967,295. They roll over to 0 on the next count. The counter inputs measure the number of rising inputs. Refer to the ***SCADAPack2 System Hardware Manual*** for further information on controller counter inputs and counter input modules.

There are three library functions related to counters. Refer to the ***Function Specification*** section for details on each function listed.

ioReadCounter5232 Read buffered data from the 5232 counter inputs.

ioReadCounter4 Read buffered data from any 4 point Counter input module.

Status LED and Output Functions

The status LED and output indicate alarm conditions. The STAT LED blinks and the STATUS output opens when an alarm occurs. The STAT LED turns off and the STATUS output closes when all alarms clear.

The STAT LED blinks a binary sequence indicating alarm codes. The sequences consist of long and short flashes, followed by an off delay of 1 second. The sequence then repeats. The sequence may be read as the Controller Status Code.

Refer to the ***SCADAPack2 System Hardware Manual*** for further information on the status LED and digital output.

There are three library functions related to the status LED and digital output. Refer to the ***Function Specification*** section for details on each function listed.

clearStatusBit Clears bits in controller status code.

getStatusBit Gets the bits in controller status code.

setStatusBit Sets the bits in controller status code.

I/O Forcing Functions

There are six library functions related to I/O forcing. Refer to the ***Function Specification*** section for details on each function listed. These functions are supported by TelePACE firmware only.

setOutputsInStopMode Sets the *doutsInStopMode* and *aoutsInStopMode* control flags to the specified state.

getOutputsInStopMode Copies the values of the output control flags into the integers pointed to by *doutsInStopMode* and *aoutsInStopMode*

clearAllForcing Removes all forcing conditions from all I/O database registers.

setForceFlag Sets the force flag(s) for the specified database register(s)

getForceFlag Copies the value of the force flag for the specified database register.

overrideDbase Writes a value to the I/O database even if the database register is currently forced

Status LED and Output Macros

The **ctools.h** file defines the following macros for use with the status LED and digital output. Refer to the ***C Tools Macros*** section for details on each macro listed.

S_MODULE_FAILURE Status LED code for I/O module communication failure

S_NORMAL Status LED code for normal status

Options Switches Functions

The controller has three option switches located under the cover of the controller module. These switches are labeled OPTION 1,2, 3 and 4. The option switches are user defined except when a SCADAPack2 I/O module or SCADAPack2 AOUT module used. In this case option switches 1 and 2 select the analog ranges. Refer to the ***SCADAPack2 System Hardware Manual*** for further information on option switches.

There is one library function related to the controller option switches. Refer to the ***Function Specification*** section for details.

optionSwitch Read option switch states.

LED Indicators Functions

An application program can control three LED indicators.

The RUN LED (green) indicates the execution status of the program. The LED can be on or off. It remains in the last state until changed.

The STAT LED (yellow) indicates error conditions. It outputs an error code as a binary sequence. The sequence repeats until a new error code is output. If the error code is zero, the status LED turns off.

The FORCE LED (yellow) indicates locked I/O variables. Use this function with caution in application programs.

There are two library functions related to the LED indicators. Refer to the ***Function Specification*** section for details on each function listed.

runLed	Controls the RUN LED status.
forceLed	Sets state of the force LED.

LED Power Control Functions

The controller board can disable the LEDs on the controller board, the upper and lower I/O modules and the 5000 Series I/O modules to conserve power. This is particularly useful in solar powered or unattended installations. Refer to the ***SCADAPack2 System Hardware Manual*** for further information on LED power control.

There are four library functions related to LED power control. Refer to the ***Function Specification*** section for details on each function listed.

ledGetDefault	Get default LED power state
ledPower	Set LED power state
ledPowerSwitch	Read LED power switch
ledSetDefault	Set default LED power state

LED Power Control Structure

The **ctools.h** file defines the structure **LED Power Control Structure** for LED power control information. Refer to the ***C Tools Structures and Types*** section for complete information on structures and enumeration types.

Software Timer Functions

The controller provides 32 powerful software timers, which greatly simplify the task of programming time-related functions. Uses include:

generation of time delays

timing of process events such as tank fill times

generation of time-based interrupts to schedule regular activities

control of digital outputs by time periods

The 32 timers are individually programmable for tick rates from ten per second to once every 25.5 seconds. Time periods from 0.1 second to greater than nineteen days can be measured and controlled.

Timer functions require an initialization step before they are used. This initialization step creates the timer support task. The function, **runTimers**, starts the timer task and must be called first in order to provide timer functionality.

There are four library functions related to timers. Refer to the ***Function Specification*** section for details on each function listed.

interval	Set timer tick interval in tenths of seconds.
settimer	Set a timer. Timers count down from the set value to zero.
timer	Read the time period remaining in a timer.
read_timer_info	Read information about a software timer.

Timer Information Structure

The **ctools.h** file defines the structure **Timer Information** for timer information. Refer to the ***C Tools Structures and Types*** section for complete information on structures and enumeration types.

Alternative Methods for Timing

If the overhead of the timer task is undesired, two alternative methods supported by the firmware exist for user timing: See the functions ***timedEvents*** and ***readStopwatch***.

Real Time Clock Functions

The controller is provided with a hardware based real time clock that independently maintains the time and date for the operating system. The time and date remain accurate during power-off. This allows the controller to be synchronized to time of day for such functions as shift production reports, automatic instrument calibration, energy logging, etc. The calendar can be used to automatically take the controller off-line during weekends and holidays. The calendar automatically handles leap years.

There are eight library functions, which access the real-time clock. Refer to the ***Function Specification*** section for details on each function listed.

alarmIn	Returns absolute time of alarm given elapsed time
getclock	Read the real time clock.
getClockAlarm	Reads the real time clock alarm settings.
getClockTime	Read the real time clock.
installClockHandler	Installs a handler for real time clock alarms.
resetClockAlarm	Resets the real time clock alarm so it will recur at the same time next day.
setclock	Set the real time clock.
setClockAlarm	Sets real time clock alarm.

Real Time Clock Structures

The **ctools.h** file defines the structures **Real Time Clock Structure** and **Alarm Settings Structure** for real time clock information. Refer to the ***C Tools Structures and Types*** section for complete information on structures and enumeration types.

Stopwatch Timer Functions

The stopwatch is a counter that increments every 10 ms. The stopwatch is useful for measuring execution times or generating delays where a fine time base is required. The stopwatch time rolls over to 0 when it reaches the maximum value for an unsigned long integer: 4,294,967,295 ms (or about 49.7 days).

There is one library function to access the stopwatch time. Refer to the ***Function Specification*** section for details.

readStopwatch	reads the stopwatch timer.

Watchdog Timer Functions

A watchdog timer is a hardware device, which enables rapid detection of computer hardware or software problems. In the event of a major problem, the CPU resets and the application program restarts.

The controller provides an integral watchdog timer to ensure reliable operation. The watchdog timer resets the CPU if it detects a problem in either the hardware or system firmware. A user program can take control of the watchdog timer, so it will detect abnormal execution of the program.

A watchdog timer is a retriggerable, time delay timer. It begins a timing sequence every time it receives a reset pulse. The time delay is adjusted so that regular reset pulses prevent the timer from expiring. If the reset pulses cease, the watchdog timer expires and turns on its output, signifying a malfunction. The timer output in the controller resets the CPU and turns off all outputs at the I/O system.

The watchdog timer is normally reset by the operating system. This is transparent to the application program. Operating in such a fashion, the watchdog timer detects any hardware or firmware problems.

The watchdog timer can detect failure of an application program. The program takes control of the timer, and resets it regularly. If unexpected operation of the program occurs, the reset pulses cease, and the watchdog timer resets the CPU. The program restarts from the beginning.

There are three library functions related to the watchdog timer. Refer to the ***Function Specification*** section for details on each function listed.

wd_auto	Gives control of the watchdog timer to the operating system (default).
wd_manual	Gives control of the watchdog timer to an application program.
wd_pulse	Generates a watchdog reset pulse.

A watchdog reset pulse must be generated at least every 500 ms. The CPU resets, and program execution starts from the beginning of the program, if the watchdog timer is not reset.

Watchdog Timer Program Example

The following program segment shows how the watchdog timer could be used to detect the failure of a section of a program.

```
wd_manual(); /* take control of watchdog timer */
do {
   /* program code */
   wd_pulse();      /* reset the watchdog timer */
}
while (condition)
wd_auto();          /* return control to OS */
```

Note: Always pass control of the watchdog timer back to the operating system before stopping a program, or switching to another task that expects the operating system to reset the timer.

Checksum Functions

To simplify the implementation of self-checking communication algorithms, the C Tools provide four types of checksums: additive, CRC-16, CRC-CCITT, and byte-wise

exclusive-OR. The CRC algorithms are particularly reliable, employing various polynomial methods to detect nearly all communication errors. Additional types of checksums are easily implemented using library functions.

There are two library functions related to checksums. Refer to the ***Function Specification*** section for details on each function listed.

checksum Calculates additive, CRC-16, CRC-CCITT and exclusive-OR type checksums

crc_reverse Calculates custom CRC type checksum using reverse CRC algorithm.

Serial Communication

The SCADAPack2 family of controllers offers three or four RS-232 serial ports. The ports are configurable for baud rate, data bits, stop bits, parity and communication protocol.

To optimize performance, minimize the length of messages on com3. Examples of recommended use for com3 are for local operator display terminals, and for programming and diagnostics using the ISaGRAF program.

Default Serial Parameters

All ports are configured at reset with default parameters when the controller is powered up in SERVICE mode. The ports use stored parameters when the controller is reset in the RUN mode. The default parameters are listed below.

Parameter	***com1***	***com2***	***Com3***	***Com4***
Baud rate	9600	9600	9600	9600
Parity	none	none	None	None
Data bits	8	8	8	8
Stop bits	1	1	1	1
Duplex	full	full	Half	full
Protocol	Modbus RTU	Modbus RTU	Modbus RTU	Modbus RTU
Addressing Mode	Standard	Standard	Standard	Standard
Station	1	1	1	1
Rx flow control	Modbus RTU	Modbus RTU	Modbus RTU	Modbus RTU
Tx flow control	none	none	none	none
Type	RS-232	RS-232	RS-232	RS-232

Debugging Serial Communication

Serial communication can be difficult to debug. This section describes the most common causes of communication failures.

To communicate, the controller and an external device must use the same communication parameters. Check the parameters in both units.

If some but not all characters transmit properly, you probably have a parity or stop bit mismatch between the devices.

The connection between two RS-232 Data Terminal Equipment (DTE) devices is made with a null-modem cable. This cable connects the transmit data output of one device to the receive data input of the other device – and vice versa. The controller is a DTE device. This cable is described in the ***System Hardware Manual*** for your controller.

The connection between a DTE device and a Data Communication Equipment (DCE) device is made with a straight cable. The transmit data output of the DTE device is

connected to the transmit data input of the DCE device. The receive data input of the DTE device is connected to the receive data output of the DCE device. Modems are usually DCE devices. This cable is described in the ***System Hardware Manual*** for your controller.

Many RS-232 devices require specific signal levels on certain pins. Communication is not possible unless the required signals are present. In the controller the CTS line must be at the proper level. The controller will not transmit if CTS is OFF. If the CTS line is not connected, the controller will force it to the proper value. If an external device controls this line, it must turn it ON for the controller to transmit.

Serial Communication Functions

The **ctools.h** file defines the following serial communication related functions. Refer to the ***Function Specification*** section for details on each function listed.

clear_errors	Clear serial port error counters.
clear_tx	Clear serial port transmit buffer.
get_port	Read serial port communication parameters.
getPortCharacteristics	Read information about features supported by a serial port.
get_status	Read serial port status and error counters.
install_handler	Install serial port character received handler.
portIndex	Get array index for serial port
portStream	Get serial port corresponding to index
queue_mode	Set serial port transmitter mode.
route	Redirect standard I/O streams.
setDTR	Control RS232 port DTR signal.
set_port	Set serial port communication parameters.

Serial Communication Structures

The **ctools.h** file defines the structures **Serial Port Configuration**, **Serial Port Status** and **Serial Port Characteristics** for serial port configuration and information. Refer to the ***C Tools Structures and Types*** section for complete information on structures and enumeration types.

Dial-Up Modem Functions

These library functions provide control of dial-up modems. They are used with external modems connected to a serial port. An external modem normally connects to the RS-232 port with a DTE to DCE cable. Consult the ***System Hardware Manual*** for your controller for details. Refer to the ***Function Specification*** section for details on each function listed.

modemInit	send initialization string to dial-up modem.
modemInitStatus	read status of modem initialization operation.
modemInitEnd	terminate modem initialization operation.
modemDial	connect with an external device using a dial-up modem.

modemDialStatus	read status of connection with external device using a dial-up modem.
modemDialEnd	terminate connection with external device using a dial-up modem.
modemAbort	unconditionally terminate connection with external device or modem initialization (used in task exit handler).
modemAbortAll	unconditionally terminate connections with external device or modem initializations (used in task exit handler).
modemNotification	notify the dial-up modem handler that an interesting event has occurred. This function is usually called whenever a message is received by a protocol.

Dial-Up Modem Macros

The **ctools.h** file defines the following macros of interest to a C application program. Refer to the ***C Tools Macros*** section for details on each macro listed.

MODEM_CMD_MAX_LEN	Maximum length of the modem initialization command string
PHONE_NUM_MAX_LEN	Maximum length of the phone number string

Dial-Up Modem Enumeration Types

The `ctools.h` file defines the enumerated types `DialError` and `DialState`. Refer to the ***C Tools Structures and Types*** section for complete information on structures and enumeration types.

Dial-up Modem Structures

The `ctools.h` file defines the structures `ModemInit` and `ModemSetup`**.** Refer to the ***C Tools Structures and Types*** section for complete information on structures and enumeration types.

Serial Communication Protocols

The TeleBUS protocols are compatible with the widely used Modbus RTU and ASCII protocols. The TeleBUS communication protocols provide a standard communication interface to SCADAPack2 controllers. Additional TeleBUS commands provide remote programming and diagnostics capability.

The TeleBUS protocols provide full access to the I/O database in the controller. The I/O database contains user-assigned registers and general purpose registers. Assigned registers map directly to the I/O hardware or system parameter in the controller. General purpose registers can be used by ladder logic and C application programs to store processed information, and to receive information from a remote device.

The TeleBUS protocols operate on a wide variety of serial data links. These include RS-232 serial ports, RS-485 serial ports, radios, leased line modems, and dial up modems. The protocols are generally independent of the communication parameters of the link, with a few exceptions.

Application programs can initiate communication with remote devices. A multiple port controller can be a data concentrator for remote devices, by polling remote devices on one port(s) and responding as a slave on another port(s).

The protocol type, communication parameters and station address are configured separately for each serial port on a controller. One controller can appear as different stations on different communication networks. The port configuration can be set from an application program, from the ISaGRAF programming software, or from another Modbus or DF1 compatible device.

Protocol Type

The protocol type may be set to emulate the Modbus ASCII and Modbus RTU protocols, or it may be disabled. When the protocol is disabled, the port functions as a normal serial port.

Station Number

The TeleBUS protocol allows up to 254 devices on a network using standard addressing and up to 65534 devices using extended addressing. Station numbers identify each device. A device responds to commands addressed to it, or to commands broadcast to all stations.

The station number is in the range 1 to 254 for standard addressing and 1 to 65534 for extended addressing. Address 0 indicates a command broadcast to all stations, and cannot be used as a station number. Each serial port may have a unique station number.

Store and Forward Messaging

Store and forward messaging allows the re-transmission of messages received by a controller communication interface. Messages may be re-transmitted on any communication interface, with or without station address translation. A user-defined translation table determines actions performed for each message. Store and forward messaging may be enabled or disabled on each port. It is disabled by default.

Serial Communication Protocol Functions

There are several library functions related to TeleBUS communication protocol. Refer to the ***Function Specification*** section for details on each function listed.

checkSFTranslationTable	Check translation table for invalid entries.
clear_protocol_status	Clears protocol message and error counters.
clearSFTranslationTable	Clear all store and forward translation table entries.
get_protocol	Reads protocol parameters.
getProtocolSettings	Reads extended addressing protocol parameters for a serial port.
get_protocol_status	Reads protocol message and error counters.
getSFTranslation	Read store and forward translation table entry.
installModbusHandler	This function allows user-defined extensions to standard Modbus protocol.
master_message	Sends a protocol message to another device.
modbusExceptionStatus	Sets response for the read exception status function.
modbusSlaveID	Sets response for the read slave ID function.
set_protocol	Sets protocol parameters and starts protocol.

setProtocolSettings	Sets extended addressing protocol parameters for a serial port.
setSFTranslation	Write store and forward translation table entry.
start_protocol	Starts protocol execution based on stored parameters.

Communication Protocols Enumeration Types

The **ctools.h** file defines the enumeration type **ADDRESS_MODE**. Refer to the ***C Tools Structures and Types*** section for complete information on structures and enumeration types.

Communication Protocols Structures

The **ctools.h** file defines the structures **Protocol Status Information**, **Protocol Settings**, **Extended Protocol Settings**, **Store and Forward Message** and **Store and Forward Status**. Refer to the ***C Tools Structures and Types*** section for complete information on structures and enumeration types.

DNP Communication Protocol

DNP, the Distributed Network Protocol, is a standards-based communications protocol developed to achieve interoperability among systems in the electric utility, oil & gas, water/waste water and security industries. This robust, flexible non-proprietary protocol is based on existing open standards to work within a variety of networks. The IEEE has recommended DNP for remote terminal unit to intelligent electronic device messaging. DNP can also be implemented in any SCADA system for efficient and reliable communications between substation computers, RTUs, IEDs and master stations; over serial or LAN-based systems.

DNP offers flexibility and functionality that go far beyond conventional communications protocols. Among its robust and flexible features DNP 3.0 includes:

- Output options
- Addressing for over 65,000 devices on a single link
- Time synchronization and time-stamped events
- Broadcast messages
- Data link and application layer confirmation

DNP 3.0 was originally designed based on three layers of the OSI seven-layer model: application layer, data link layer and physical layer. The application layer is object-based with objects provided for most generic data formats. The data link layer provides for several methods of retrieving data such as polling for classes and object variations. The physical layer defines most commonly a simple RS-232 or RS-485 interface.

DNP Communication Protocol Functions

There are several library functions related to DNP communication protocol. Refer to the ***Function Specification*** section for details on each function listed.

dnpClearEventLogs	Deletes all change events from the DNP change event buffers.
dnpConnectionEvent	Report a DNP connection event

dnpCreateAddressMappingTable	Allocates memory for a new address mapping table according to the 'size' parameter.
dnpCreateMasterPollTable	Allocates memory for a new table according to the 'size' parameter.
dnpCreateRoutingTable	Allocates memory for a new routing table according to the 'size' parameter.
dnpGenerateChangeEvent	Generates a change event for the DNP point.
dnpGenerateEventLog	Generates a change event for the DNP point.
dnpGetAI16Config	Reads the configuration of a DNP 16-bit analog input point.
dnpGetAI32Config	Reads the configuration of a DNP 32-bit analog input point.
dnpGetAISFConfig	Reads the configuration of a DNP 32-bit short floating analog input point.
dnpGetAO16Config	Reads the configuration of a DNP 16-bit analog output point.
dnpGetAO32Config	Reads the configuration of a DNP 32-bit analog output point.
dnpGetAOSFConfig	Sets the configuration of a DNP 32-bit short floating analog output point.
dnpGetCI16Config	Reads the configuration of a DNP 16-bit counter input point.
dnpGetCI32Config	Reads the configuration of a DNP 32-bit counter input point.
dnpGetBIConfig	Reads the configuration of a DNP binary input point.
dnpGetBIConfigEx	Reads the configuration of an extended DNP Binary Input point.
dnpGetBOConfig	Reads the configuration of a DNP binary output point.
dnpGetCI16Config	Reads the configuration of a DNP 16-bit counter input point.
dnpGetCI32Config	Reads the configuration of a DNP 32-bit counter input point.
dnpGetConfiguration	Reads the DNP protocol configuration.
dnpGetConfigurationEx	Reads the extended DNP configuration parameters.
dnpGetRuntimeStatus	Reads the current status of all DNP change event buffers.
dnpInstallConnectionHandler	Configures the connection handler for DNP.

dnpMasterClassPoll	Sends a Class Poll message in DNP, to request the specified data classes from a DNP slave.
DnpMasterClockSync	sends a Clock Synchronization message in DNP, to a DNP slave.
dnpPortStatus	Returns the DNP message statistics for the specified communication port.
dnpReadAddressMappingTableEntry	Reads an entry from the DNP address mapping table.
dnpReadAddressMappingTableSize	Reads the total number of entries in the DNP address mapping table.
dnpReadMasterPollTableEntry	Reads an entry from the DNP master poll table.
dnpReadMasterPollTableEntryEx	Reads an extended entry from the DNP master poll table.
dnpReadPMasterPollTableSize	Reads the total number of entries in the DNP master poll table.
dnpReadRoutingTableEntry	Reads an entry from the routing table.
dnpReadRoutingTableEntryEx	Reads an extended entry from the DNP routing table.
dnpReadRoutingTableEntry_DialString	Reads a primary and secondary dial string from an entry in the DNP routing table.
dnpReadRoutingTableSize	Reads the total number of entries in the routing table.
dnpSaveAI16Config	Sets the configuration of a DNP 16-bit analog input point.
dnpSaveAI32Config	Sets the configuration of a DNP 32-bit analog input point.
dnpSaveAISFConfig	Sets the configuration of a DNP 32-bit short floating analog input point
dnpSaveAO16Config	Sets the configuration of a DNP 32-bit analog output point.
dnpSaveAO32Config	Sets the configuration of a DNP 32-bit analog output point.
dnpSaveAOSFConfig	Sets the configuration of a DNP 32-bit short floating analog output point.
dnpSaveBIConfig	Sets the configuration of a DNP binary input point.
dnpSaveBOConfig	Sets the configuration of a DNP binary output point.
dnpSaveCI16Config	Sets the configuration of a DNP 16-bit counter input point.
dnpSaveCI32Config	Sets the configuration of a DNP 32-bit counter input point.

dnpSaveConfiguration	Defines DNP protocol configuration parameters.
dnpSaveConfigurationEx	Writes the extended DNP configuration parameters
dnpSendUnsolicitedResponse	Sends an 'Unsolicited Response' message in DNP protocol.
dnpSearchRoutingTable	Searches the routing table for a specific DNP address.
dnpStationStatus	Returns the DNP message statistics for a remote DNP station.
dnpWriteAddressMappingTableEntry	Writes an entry in the DNP address mapping table.
dnpWriteMasterApplicationLayerConfig	Writes DNP Master application layer configuration.
dnpWriteMasterPollTableEntry	Writes an entry in the DNP master poll table.
dnpWriteRoutingTableEntry	Writes an entry in the DNP routing table.
dnpWriteRoutingTableEntryEx	Writes an extended entry in the DNP routing table.
dnpWriteRoutingTableEntry_DialString	Writes a primary and secondary dial string into an entry in the DNP routin

DNP Communication Protocol Structures and Types

The ctools.h file defines the structures DNP Configuration, Binary Input Point, Binary Output Point, Analog Input Point, Analog Output Point and Counter Input Point. Refer to the *C Tools Structures and Types* section for complete information on structures and enumeration types.

PPP Communication Protocol

PPP, the Point-to-Point Network Protocol, is a standards-based communications protocol developed to achieve interoperability among systems.

PPP Communication Protocol Functions

There are several library functions related to PPP communication protocol. Refer to the ***Function Specification*** section for details on each function listed.

pppGetInterfaceHandle	Returns PPP interface handle for the specified serial port.
pppReadSettings	Reads the PPP settings for the specified serial port.
pppReadUserTableEntry	Reads the entry at *`index`* from the PPP username table.
pppReadUserTableSize	Returns the number of entries in the PPP username table.
pppWriteSettings	Writes the PPP settings for the specified serial port.
pppWriteUserTableEntry	Writes an entry at *`index`* into the PPP username table.
pppWriteUserTableSize	Writes the size of the PPP username table.

PPP Communication Protocol Structures and Types

The ctools.h file defines the structure PPP_LOGIN_TYPE and enumerated type PPP_STRUCTURE. Refer to the *C Tools Structures and Types* section for complete information on structures and enumeration types.

DF1 Communication Protocol

The TeleBUS DF1 protocol supports the DF1 Basic Command Set in the Half Duplex and Full Duplex DF1 protocols.

DF1 Communication Protocol Functions

There are several library functions related to DF1 communication protocol. Refer to the ***Function Specification*** section for details on each function listed.

getABConfiguration	Reads DF1 protocol configuration parameters.
pollABSlave	Requests a response from a slave controller using the half-duplex version of the protocol.
resetAllABSlaves	Clears responses from the response buffers of half-duplex slave controllers.
setABConfiguration	Defines DF1 protocol configuration parameters.

TCP/IP Communications

The SCADAPack2 and SCADAPack2P controllers have one 10BaseT Ethernet port. 10BaseT is a single communications channel running at 10MHz over unshielded, twisted - pair cabling.

TCP/IP Functions

The **ctools.h** file defines the following TCP/IP related functions. Refer to the ***Function Specification*** section for details on each function listed.

ethernetGetIP	Get the Ethernet controller TCP/IP settings.
ethernetSetIP	Set the Ethernet controller TCP/IP settings.
ethernetGetMACAddress	Returns Ethernet controller MAC address.
ipGetConnectionSummary	Returns the number of connections: master, slave or unused.
ipGetInterfaceType	Returns the interface that is configured to the specified local IP address.

Modbus IP Protocol

Modbus IP is an extension of serial Modbus, which defines how Modbus messages are encoded within and transported over TCP/IP-based networks. Modbus IP protocols are just as simple to implement and flexible to apply as serial Modbus. Complete information for Modbus IP and serial Modbus may be found on-line at www.modbus.org/.

Modbus IP Functions

The **ctools.h** file defines the following Modbus IP related functions. Refer to the ***Function Specification*** section for details on each function listed.

mTcpSetConfig	Set Modbus IP protocol settings.
mTcpGetConfig	Get Modbus IP protocol settings.
mTcpSetInterface	Set interface settings used by the Modbus IP protocols.
mTcpGetInterface	Get interface settings used by the Modbus IP protocols.
mTcpSetInterfaceEx	Set interface settings used by the Modbus IP protocols including Enron Modbus settings.
mTcpGetInterfaceEx	Get interface settings used by the Modbus IP protocols including Enron Modbus settings.
mTcpSetProtocol	Get interface settings used by the Modbus IP protocols.
mTcpGetProtocol	Get interface settings used by the Modbus IP protocols.
mTcpMasterOpen	Allocates a connection ID and creates a task to service a Modbus IP master messaging connection.
mTcpMasterMessage	Builds the Modbus command and sends a message to the mastering task to tell it to send the command.
mTcpMasterStatus	Returns the master command status for the specified connection.
mTcpMasterDisconnect	Tells a Modbus IP master task to disconnect and end the task.
mTcpMasterClose	Returns a master connection ID to the connection pool.

Sockets API

These functions provide support for the BSD 4.4 Socket API. Additional Socket Extension functions are also provided. These apply specifically to the SCADPack32 TCP/IP Stack.

Refer to the ***Function Specification*** section for details on each function listed.

accept	**tfIoctl**
bind	**tfRead**
connect	**tfWrite**
getpeername	**writev**
getsockname	**tfBindNoCheck**
getsockopt	**tfBlockingState**
htonl	**tfFreeZeroCopyBuffer**
htons	**tfGetOobDataOffset**
inet_addr	**tfGetSocketError**
inet_aton	**tfGetSendCompltBytes**
listen	**tfGetWaitingBytes**
ntohl	**tfGetZeroCopyBuffer**
ntohs	**tfInetToAscii**

readv
recv
recvfrom
rresvport
select
send
sendto
setsockopt
shutdown
socket
tfClose
tfGetPppDnsIpAddress
tfPppSetOption
tfUseDialer
tfDialerAddExpectSend
tfResetConnection
tfSocketArrayWalk
tfZeroCopyRecv
tfZeroCopyRecvFrom
tfZeroCopySend
tfZeroCopySendTo
tfRegisterSocketCB
tfRegisterSocketCBParam
tfPingOpenStart
tfPingClose
tfPingGetStatistics
tfGetPppPeerIpAddress
tfSetPppPeerIpAddress
tfDialerAddSendExpect

Modbus I/O Database

The Modbus database is a user-defined database that allows data to be shared between TelePACE or ISaGRAF programs, C++ programs and communication protocols.

TelePACE and ISaGRAF firmware support different ranges of Modbus Database registers. The following table shows the register ranges for these firmware types.

TelePACE Modbus Addresses	ISaGRAF Modbus Addresses	Data Type
00001 to 04096	00001 to 09999	Coil Register 1 returned if variable is non-zero; 0 returned if variable is 0
10001 to 14096	10001 to 19999	Status Register 1 returned if variable is non-zero; 0 returned if variable is 0
30001 to 39999	30001 to 39999	Input Register word (16 bits)
40001 to 49999	40001 to 49999	Holding Register word (16 bits)

Modbus I/O Database Register Types

The I/O database is divided into four types of I/O registers. Each of these types is initially configured as general purpose registers by the controller.

Coil Registers

Coil, or digital output, database registers may be assigned to 5000 Series digital output modules or SCADAPack I/O modules through the Register Assignment. Coil registers

may also be assigned to controller on-board digital outputs and to system configuration modules.

Status Registers

Status, or digital input, database registers may be assigned to 5000 Series digital input modules or SCADAPack I/O modules through the Register Assignment. Status registers may also be assigned to controller on-board digital inputs and to system diagnostic modules.

Input Registers

Input, or analog input, database registers may be assigned to 5000 Series analog input modules or SCADAPack I/O modules through the Register Assignment. Input registers may also be assigned to controller internal analog inputs and to system diagnostic modules.

Holding Registers

Holding, or analog output, database registers may be assigned to 5000 Series analog output modules or SCADAPack analog output modules through the Register Assignment. Holding registers may also be assigned to system diagnostic and configuration modules.

Modbus I/O Database Functions

There are several library functions related to the Modbus database. Refer to the ***Function Specification*** section for details on each function listed.

dbase	Reads a value from the database.
installDbaseHandler	Allows an extension to be defined for the **dbase** function.
installSetdbaseHandler	Allows an extension to be defined for the **setdbase** function.
Dbase Handler Function	User-defined function that handles reading of Modbus addresses not assigned in the ISaGRAF Dictionary.
setdbase	W**r**ites a value to the database.
Setdbase Handler Function	User-defined function that handles writing to Modbus addresses not assigned in the ISaGRAF Dictionary.

Modbus I/O Database Macros

The **ctools.h** file defines library functions for the I/O database. Refer to the ***C Tools Macros*** section for details on each macro listed.

AB	Specifies Allan-Bradley database addressing.
DB_BADSIZE	Error code: out of range address specified
DB_BADTYPE	Error code: bad database addressing type specified
DB_OK	Error code: no error occurred
LINEAR	Specifies linear database addressing.
MODBUS	Specifies Modbus database addressing.
NUMAB	Number of registers in the Allan-Bradley database.

NUMCOIL	Number of registers in the Modbus coil section.
NUMHOLDING	Number of registers in the Modbus holding register section.
NUMINPUT	Number of registers in the Modbus input registers section.
NUMLINEAR	Number of registers in the linear database.
NUMSTATUS	Number of registers in the Modbus status section.
START_COIL	Start of the coil section in the linear database.
START_HOLDING	Start of the holding registers section in the linear database.
START_INPUT	Start of the input register section in the linear database.
START_STATUS	Start of the status section in the linear database.

Register Assignment

All I/O hardware that is used by the controller must be assigned to I/O database registers in order for these I/O points to be scanned continuously. I/O data may then be accessed through the I/O database within the C program. C programs may read data from, or write data to the I/O hardware through user- assigned registers in the I/O database.

The Register Assignment assigns I/O database registers to user-assigned registers using I/O modules. An I/O Module can refer to an actual I/O hardware module (e.g. *5401 Digital Input Module*) or it may refer to a set of controller parameters, such as serial port settings.

The chapter *Register Assignment Reference* of the ***TelePACE Ladder Logic Reference and User Manual*** contains a description of what each module is used for and the register assignment requirements for the I/O module.

Register assignments configured using the TelePACE *Register Assignment* dialog may be stored in the TelePACE program file or downloaded directly to the controller. To obtain error checking that prevents invalid register assignments, use the *TelePACE Register Assignment* dialog to initially build the Register Assignment. The Register Assignment can then be saved in a Ladder Logic file (e.g. filename.lad) and downloaded with the C program.

Register Assignment Functions

There are several library functions related to register assignment. Refer to the ***Function Specification*** section for details on each function listed.

clearRegAssignment	Erases the current Register Assignment.
addRegAssignment	Adds one I/O module to the current Register Assignment.
getIOErrorIndication	Gets the control flag for the I/O module error indication
getOutputsInStopMode	Gets the control flags for state of Outputs in Ladders Stop Mode
setIOErrorIndication	Sets the control flag for the I/O module error indication
setOutputsInStopMode	Sets the control flags for state of Outputs in Ladders Stop Mode

Register Assignment Enumeration Types

The **ctools.h** file defines one enumeration type. The **ioModules** enumeration type defines a list of results of sending a command. Refer to the ***C Tools Structures and Types*** section for complete information on structures and enumeration types.

Register Assignment Structure

The **ctools.h** file defines the structure **RegAssign.** Refer to the ***C Tools Structures and Types*** section for complete information on structures and enumeration types.

ISaGRAF Variable Access Functions

Variables declared in an ISaGRAF application are accessed from a C application using the ISaGRAF variable access functions listed below. Refer to the ***Function Specification*** section for details on each function listed.

readBoolVariable	Returns the current value of the specified boolean variable.
readIntVariable	Returns the current value of the specified integer variable.
readRealVariable	Returns the current value of the specified real variable.
readMsgVariable	Returns the current value of the specified message variable.
readTimerVariable	Returns the current value of the specified timer variable.
writeBoolVariable	Writes to the specified boolean variable.
writeIntVariable	Writes to the specified integer variable.
writeRealVariable	Writes to the specified real variable.
writeMsgVariable	Writes to the specified message variable.
writeTimerVariable	Writes to the specified timer variable.

HART Communication

The HART ® protocol is a field bus protocol for communication with smart transmitters.

The HART protocol driver provides communication between TeleSAFE Micro16 and SCADAPack2 controllers and HART devices. The protocol driver uses the model 5904 HART modem for communication. Four HART modem modules are supported per controller.

The driver allows HART transmitters to be used with C application programs and with RealFLO. The driver can read data from HART devices.

HART Command Functions

The **ctools.h** file defines the following HART command related functions. Refer to the ***Function Specification*** section for details on each function listed.

hartIO	Reads data from the 5904 interface module, processes HART responses, processes HART commands, and writes commands and configuration data to the 5904 interface module.
hartCommand	send a HART command string and specify a function to handle the response

hartCommand0	read unique identifier using short-address algorithm
hartCommand1	read primary variable
hartCommand2	read primary variable current and percent of span
hartCommand3	read primary variable current and dynamic variables
hartCommand11	read unique identifier associated with tag
hartCommand33	read specified transmitter variables
hartStatus	return status of last HART command sent
hartGetConfiguration	read HART module settings
hartSetConfiguration	write HART module settings
hartPackString	convert string to HART packed string
hartUnpackString	convert HART packed string to string

HART Command Macros

The **ctools.h** file defines the following macro of interest to a C application program. Refer to the ***C Tools Macros*** section for details.

DATA_SIZE	Maximum length of the HART command or response field.

HART Command Enumeration Types

The **ctools.h** file defines one enumeration type. The **HART_RESULT** enumeration type defines a list of results of sending a command. Refer to the ***C Tools Structures and Types*** section for complete information on structures and enumeration types.

HART Command Structures

The **ctools.h** file defines five structures. Refer to the ***C Tools Structures and Types*** section for complete information on structures and enumeration types.

The HART_DEVICE type is a structure containing information about the HART device.

The HART_VARIABLE type is a structure containing a variable read from a HART device.

The HART_SETTINGS type is a structure containing the configuration for the HART modem module.

The HART_COMMAND type is a structure containing a command to be sent to a HART slave device.

The HART_RESPONSE type is a structure containing a response from a HART slave device.

Function Specifications

This section of the user manual contains specifications for using each of the available functions. The functions in the sections that follow are available for use in C++ programs. These functions are available for use with both TelePACE and ISaGRAF firmware unless otherwise noted.

Functions Supported by TelePACE Only

The following functions are only supported by C++ Tools running on TelePACE firmware:

- addRegAssignment
- clearRegAssignment
- getForceFlag
- getOutputsInStopMode
- overrideDbase
- setForceFlag
- setOutputsInStopMode

Functions Supported by ISaGRAF Only

The following functions are only supported by C++ Tools running on ISaGRAF firmware:

- Dbase Handler Function
- installDbaseHandler
- installSetdbaseHandler
- readBoolVariable
- readIntVariable
- readMsgVariable
- readRealVariable
- readTimerVariable
- read_timer_info
- Setdbase Handler Function
- writeBoolVariable
- writeIntVariable
- writeMsgVariable
- writeRealVariable
- writeTimerVariable

accept

Syntax

```
include <ctools.h>
int accept
(
int socketDescriptor,
struct sockaddr * addressPtr,
int * addressLengthPtr
);
```

Function Description

The argument *socketDescriptor* is a socket that has been created with **socket,** bound to an address with **bind**, and that is listening for connections after a call to **listen**. **accept** extracts the first connection on the queue of pending connections, creates a new socket with the properties of *socketDescriptor*, and allocates a new socket descriptor for the socket. If no pending connections are present on the queue and the socket is not marked as non-blocking, **accept** blocks the caller until a connection is present. If the socket is marked as non-blocking and no pending connections are present on the queue, **accept** returns an error as described below. The accepted socket is used to **send** and **recv** data to and from the socket that it is connected to. It is not used to accept more connections. The original socket remains open for accepting further connections. **accept** is used with connection-based socket types, currently with SOCK_STREAM.

Using **select** (prior to calling **accept**):

It is possible to **select** a listening socket for the purpose of an **accept** by selecting it for a read. However, this will only indicate when a connect indication is pending; it is still necessary to call **accep**t.

Parameters

socketDescriptor	The socket descriptor that was created with **socket** and bound to with **bind** and is listening for connections with **listen.**
addressPtr	The structure to write the incoming address into.
addressLengthPtr	Initially, it contains the amount of space pointed to by *addressPtr.* On return it contains the length in bytes of the address returned.

Returns

New Socket Descriptor or –1 on error.

If **accept** fails, the errorCode can be retrieved with **getErrorCode**(socketDescriptor) which will return one of the following error codes:

EBADF	The socket descriptor is invalid.
EINVAL	addressPtr was a null pointer.
EINVAL	addressLengthPtr was a null pointer.
EINVAL	The value of addressLengthPtr was too small.
ENOBUFS	There was insufficient user memory available to complete the operation.

EPERM	Cannot call **accept** without calling **listen** first.
EOPNOTSUPP	The referenced socket is not of type SOCK_STREAM.
EPROTO	A protocol error has occurred; for example, the connection has already been released.
EWOULDBLOCK	The socket is marked as non-blocking and no connections are present to be accepted.

addRegAssignment

Add Register Assignment (TelePACE firmware only)

Syntax

```
#include <ctools.h>
BOOLEAN addRegAssignment(
      UINT16 moduleType,
      INT16 moduleAddress,
      UINT16 startingRegister1,
      UINT16 startingRegister2,
      UINT16 startingRegister3,
      UINT16 startingRegister4);
```

Description

The **addRegAssignment** function adds one I/O module to the current Register Assignment of type *moduleType*. The following symbolic constants are valid values for *moduleType*:

```
AIN_520xTemperature          CNTR_520xCounterInputs
AIN_520xRAMBattery           CNTR_5410
AIN_5501                     DIAG_commStatus
AIN_5502                     DIAG_controllerStatus
AIN_5503                     DIAG_forceLED
AIN_5504                     DIAG_IPConnections
AIN_5505                     DIAG_ModbusStatus
AIN_5506                     DIAG_protocolStatus
AIN_5521                     DIN_5401
AIN_generic8                 DIN_5402
AOUT_5301                    DIN_5403
AOUT_5302                    DIN_5404
AOUT_5304                    DIN_5405
AOUT_generic2                DIN_5421
AOUT_generic4                DIN_generic16
CNFG_5904Modem               DIN_generic8
CNFG_clearPortCounters       DOUT_5401
CNFG_clearProtocolCounters   DOUT_5402
CNFG_IPSettings              DOUT_5406
CNFG_LEDPower                DOUT_5407
CNFG_modbusIpProtocol        DOUT_5408
CNFG_MTCPIfSettings          DOUT_5409
CNFG_MTCPSettings            DOUT_5411
CNFG_PIDBlock                DOUT_generic16
CNFG_portSettings            DOUT_generic8
CNFG_protocolExtended        SCADAPack_AOUT
CNFG_protocolExtendedEx      SCADAPack_lowerIO
CNFG_protocolSettings        SCADAPack_upperIO
CNFG_realTimeClock           SCADAPack_LPIO
CNFG_saveToEEPROM            SCADAPack_2IO
CNFG_setSerialPortDTR        SCADAPack_100IO
CNFG_storeAndForward         SCADAPack_5606IO
```

moduleAddress specifies a unique address for the module. For the valid range for *moduleAddress* refer to the list of modules in the chapter *Register Assignment Reference* of the ***TelePACE Ladder Logic Reference and User Manual***. For module addresses

com1, com2, com3 or com4 specify 0, 1, 2 or 3 respectively for *moduleAddress*. For module address Ethernet1 specify 4 for *moduleAddress*. For module types that have no module address (e.g. CNFG_LEDPower) specify -1 for *moduleAddress.* For SCADAPack module types that have a module address fixed at 0, specify 0 for *moduleAddress.*

startingRegister1 specifies the first register of any unused block of consecutive registers. Refer to the list of modules in the *Register Assignment Reference* for the type and number of registers required for this block. Data read from or written to the module is stored in this block of registers.

If the module type specified has more than one type of I/O, use *startingRegister2*, *startingRegister3*, and *startingRegister4* as applicable. Each start register specifies the first register of an unused block of consecutive registers for each type of input or output on the module. Refer to the list of modules in the *Register Assignment Reference* for the module I/O types. Specify 0 for *startingRegister2*, *startingRegister3*, or *startingRegister4* if not applicable.

Notes

Up to 150 modules may be added to the Register Assignment. If the Register Assignment is full or if an incorrect value is specified for any argument this function returns FALSE; otherwise TRUE is returned.

Output registers specified for certain CNFG type modules are initialized with the current parameter values when the module is added to the Register Assignment (e.g. CNFG_realTimeClock).

Call **clearRegAssignment** first before using the **addRegAssignment** function when creating a new Register Assignment.

Duplicate or overlapping register assignments are not checked for by this function. Overlapping register assignments may result in unpredictable I/O activity.

To obtain error checking that prevents invalid register assignments such as these, use the *TelePACE Register Assignment* dialog to build the Register Assignment. Then save the Register Assignment in a Ladder Logic file (e.g. `filename.lad`) and download it with the C program, or transfer the Register Assignment to the C program using the **clearRegAssignment** and **addRegAssignment** functions.

To save the Register Assignment with the controller settings in flash memory so that it is loaded on controller reset, call **flashSettingsSave** as shown in the example below.

The IO_SYSTEM resource must be requested before calling this function.

See Also

clearRegAssignment

Example

```
#include <ctools.h>

int main(void)
{
       request_resource(IO_SYSTEM);

       /* Create the Register Assignment */
       clearRegAssignment();

       addRegAssignment(SCADAPack_2IO, 0, 1,
          10001, 30001, 40001);
```

```
        addRegAssignment(AOUT_5302, 1, 40003, 0, 0, 0);
        addRegAssignment(DIAG_forceLED, -1, 10017, 0, 0, 0);
        addRegAssignment(DIAG_controllerStatus, -1, 30009, 0, 0, 0);
        addRegAssignment(DIAG_protocolStatus, 2, 30010, 0, 0, 0);

        release_resource(IO_SYSTEM);

        // save register assignment with controller settings

        request_resource(FLASH_MEMORY);
        flashSettingsSave(CS_PERMANENT);
        release_resource(FLASH_MEMORY);
}
```

addRegAssignmentEx

Add Register Assignment (TelePACE firmware only)

Syntax

```
#include <ctools.h>
BOOLEAN addRegAssignmentEx(
      UINT16 moduleType,
      INT16 moduleAddress,
      UINT16 startingRegister1,
      UINT16 startingRegister2,
      UINT16 startingRegister3,
      UINT16 startingRegister4,
      UINT16 parameters[16]
);
```

Description

The addRegAssignmentEx function adds one I/O module to the current Register Assignment of type *moduleType*. The following symbolic constants are valid values for *moduleType*:

```
AIN_520xTemperature          CNTR_520xCounterInputs
AIN_520xRAMBattery           CNTR_5410
AIN_5501                     DIAG_commStatus
AIN_5502                     DIAG_controllerStatus
AIN_5503                     DIAG_forceLED
AIN_5504                     DIAG_IPConnections
AIN_5505                     DIAG_ModbusStatus
AIN_5506                     DIAG_protocolStatus
AIN_5521                     DIN_5401
AIN_generic8                 DIN_5402
AOUT_5301                    DIN_5403
AOUT_5302                    DIN_5404
AOUT_5304                    DIN_5405
AOUT_generic2                DIN_5421
AOUT_generic4                DIN_generic16
CNFG_5904Modem               DIN_generic8
CNFG_clearPortCounters       DOUT_5401
CNFG_clearProtocolCounters   DOUT_5402
CNFG_IPSettings              DOUT_5406
CNFG_LEDPower                DOUT_5407
CNFG_modbusIpProtocol        DOUT_5408
CNFG_MTCPIfSettings          DOUT_5409
CNFG_MTCPSettings            DOUT_5411
CNFG_PIDBlock                DOUT_generic16
CNFG_portSettings            DOUT_generic8
CNFG_protocolExtended        SCADAPack_AOUT
CNFG_protocolExtendedEx      SCADAPack_lowerIO
CNFG_protocolSettings        SCADAPack_upperIO
CNFG_realTimeClock           SCADAPack_LPIO
CNFG_saveToEEPROM            SCADAPack_2IO
CNFG_setSerialPortDTR        SCADAPack_100IO
CNFG_storeAndForward         SCADAPack_5606IO
```

moduleAddress specifies a unique address for the module. For the valid range for *moduleAddress* refer to the list of modules in the chapter *Register Assignment Reference*

of the ***TelePACE Ladder Logic Reference and User Manual***. For module addresses com1, com2, com3 or com4 specify 0, 1, 2 or 3 respectively for *moduleAddress*. For module address Ethernet1 specify 4 for *moduleAddress*. For module types that have no module address (e.g. CNFG_LEDPower) specify -1 for *moduleAddress.* For SCADAPack module types that have a module address fixed at 0, specify 0 for *moduleAddress.*

startingRegister1 specifies the first register of any unused block of consecutive registers. Refer to the list of modules in the *Register Assignment Reference* for the type and number of registers required for this block. Data read from or written to the module is stored in this block of registers.

If the module type specified has more than one type of I/O, use *startingRegister2*, *startingRegister3*, and *startingRegister4* as applicable. Each start register specifies the first register of an unused block of consecutive registers for each type of input or output on the module. Refer to the list of modules in the *Register Assignment Reference* for the module I/O types. Specify 0 for *startingRegister2*, *startingRegister3*, or *startingRegister4* if not applicable.

`parameters` is an array of configuration parameters for the register assignment module. Most modules do not use the parameters. Use the `addRegAssignment` function to configure these modules. Use parameters with the following modules.

5505 I/O Module: `parameters[0]` to `[3]` define the analog input type for the corresponding input. Valid values are:

- 0 = RTD in deg Celsius
- 1 = RTD in deg Fahrenheit
- 2 = RTD in deg Kelvin
- 3 = resistance measurement in ohms.

5505 I/O Module: `parameters[4]` defines the analog input filter. Valid values are:

- 0 = 0.5 s
- 1 = 1 s
- 2 = 2 s
- 3 = 4 s

5506 I/O Module: `parameters[0]` to `[7]` define the analog input type for the corresponding input. Valid values are:

- 0 = 0 to 5 V input
- 1 = 1 to 5 V input
- 2 = 0 to 20 mA input
- 3 = 4 to 20 mA input

5506 I/O Module: `parameters[8]` defines the analog input filter. Valid values are:

- 0 = < 3 Hz (maximum filter)
- 1 = 6 Hz
- 2 = 11 Hz
- 3 = 30 Hz (minimum filter)

5506 I/O Module: `parameters[9]` defines the scan frequency. Valid values are:

- 0 = 60 Hz
- 1 = 50 Hz

5606 I/O Module: `parameters[0]` to `[7]` define the analog input type for the corresponding input. Valid values are:

- 0 = 0 to 5 V input

- 1 = 1 to 5 V input
- 2 = 0 to 20 mA input
- 3 = 4 to 20 mA input

5606 I/O Module: `parameters[8]` defines the analog input filter. Valid values are:

- 0 = < 3 Hz (maximum filter)
- 1 = 6 Hz
- 2 = 11 Hz
- 3 = 30 Hz (minimum filter)

5606 I/O Module: `parameters[9]` defines the scan frequency. Valid values are:

- 0 = 60 Hz
- 1 = 50 Hz

5606 I/O Module: `parameters[10]` defines the analog output type. Valid values are:

- 0 = 0 to 20 mA output
- 1 = 4 to 20 mA output

Notes

Up to 150 modules may be added to the Register Assignment. If the Register Assignment is full or if an incorrect value is specified for any argument this function returns FALSE; otherwise `TRUE` is returned.

Output registers specified for certain CNFG type modules are initialized with the current parameter values when the module is added to the Register Assignment (e.g. `CNFG_realTimeClock`).

Call `clearRegAssignment` first before using the `addRegAssignmentEx` function when creating a new Register Assignment.

Duplicate or overlapping register assignments are not checked for by this function. Overlapping register assignments may result in unpredictable I/O activity.

To obtain error checking that prevents invalid register assignments such as these, use the *TelePACE Register Assignment* dialog to build the Register Assignment. Then save the Register Assignment in a Ladder Logic file (e.g. `filename.lad`) and download it with the C program, or transfer the Register Assignment to the C program using the `clearRegAssignment` and `addRegAssignmentEx` functions.

To save the Register Assignment with the controller settings in flash memory so that it is loaded on controller reset, call `flashSettingsSave` as shown in the example below.

The `IO_SYSTEM` resource must be requested before calling this function.

See Also

`addRegAssignment, clearRegAssignment`

Example

```
#include <ctools.h>

int main(void)
{
       UINT16 parameters[16];

       request_resource(IO_SYSTEM);
```

```
        /* Create the Register Assignment */
        clearRegAssignment();

        /* add a 5606 module */
        parameters[0] = 0;  // 0 to 5 V
        parameters[1] = 0;  // 0 to 5 V
        parameters[2] = 0;  // 0 to 5 V
        parameters[3] = 0;  // 0 to 5 V
        parameters[4] = 3;  // 4 to 20 mA
        parameters[5] = 3;  // 4 to 20 mA
        parameters[6] = 3;  // 4 to 20 mA
        parameters[7] = 3;  // 4 to 20 mA
        parameters[8] = 0;  // 3 Hz input filter
        parameters[9] = 0;  // 60 Hz scan frequency
        parameters[10] = 1; // 4 to 20 mA outputs

        addRegAssignmentEx(SCADAPack_5606IO, 0, 1, 10001, 30001, 40001,
parameters);
        release_resource(IO_SYSTEM);

        // save register assignment with controller settings
        request_resource(FLASH_MEMORY);
        flashSettingsSave(CS_PERMANENT);
        release_resource(FLASH_MEMORY);
}
```

alarmIn

Determine Alarm Time from Elapsed Time

Syntax

```
#include <ctools.h>
ALARM_SETTING alarmIn(UINT16 hours, UINT16 minutes, UINT16 seconds);
```

Description

The **alarmIn** function calculates the alarm settings to configure a real time clock alarm to occur in *hours*, *minutes* and *seconds* from the current time.

The function returns an ALARM_SETTING structure suitable for passing to the **setClockAlarm** function. The structure specifies an absolute time alarm at the time offset specified by the call to **alarmIn**. Refer to the **Structures and Types** section for a description of the fields in the ALARM_SETTING structure.

Notes

If *second* is greater than 60 seconds, the additional time is rolled into the minutes. If *minute* is greater than 60 minutes, the additional time is rolled into the hours.

If the offset time is greater that one day, then the alarm time will roll over within the current day.

The IO_SYSTEM resource must be requested before calling this function.

See Also

setClockAlarm

allocate_envelope

Obtain an Envelope from the RTOS

Syntax

```
#include <ctools.h>
envelope *allocate_envelope(void);
```

Description

The **allocate_envelope** function obtains an envelope from the operating system. If no envelope is available, the task is blocked until one becomes available.

The **allocate_envelope** function returns a pointer to the envelope.

Notes

Envelopes are used to send messages between tasks. The RTOS allocates envelopes from a pool of free envelopes. It returns envelopes to the pool when they are de-allocated.

An application program must ensure that unneeded envelopes are de-allocated. Envelopes may be reused.

See Also

deallocate_envelope

Example

```
#include <ctools.h>
extern UINT32 other_task_id;

void task1(void)
{
       envelope *letter;

       /* send a message to another task */
       /* assume it will deallocate the envelope */

       letter = allocate_envelope();
       letter->destination = other_task_id;
       letter->type = MSG_DATA;
       letter->data = 5;
       send_message(letter);

       /* receive a message from any other task */

       letter = receive_message();
       /* ... process the data here */
       deallocate_envelope(letter);

       /* ... the rest of the task */
}
```

allocateMemory

Allocate Non-Volatile Dynamic Memory

Syntax

```
#include <ctools.h>
BOOLEAN allocateMemory(void **ppMemory, UINT32 size)
```

Description

The **allocateMemory** function allocates the requested memory from the system memory pool. The pool is a separate area of memory from the system heap. Memory in the system pool is preserved when the controller is reset.

The function has two arguments: *ppMemory,* a pointer to a pointer to the memory allocated; and *size,* the number of bytes of memory to be allocated.

The function returns TRUE if the memory was allocated and FALSE if the memory is not available.

Use the `freeMemory` function to free non-volatile memory.

Notes

The DYNAMIC_MEMORY resource must be requested before calling this function.

The allocation of memory and the allocated memory are non-volatile.

Pointers to non-volatile dynamic memory must be statically allocated in a non-volatile data section. Otherwise they will be initialised at reset and the non-volatile dynamic memory will be lost. The example below demonstrates how to create a non-volatile data section to save pointers to non-volatile dynamic memory.

See Also

freeMemory

Example

See the Memory Allocation Example in the Examples section.

bind

Bind an address to an unnamed socket

Syntax

```
#include <ctools.h>
int bind(
       int socketDescriptor,
       const struct sockaddr * addressPtr,
       int addressLength);
```

Function Description

bind assigns an address to an unnamed socket. When a socket is created with **socket**, it exists in an address family space but has no address assigned. **bind** requests that the address pointed to by *addressPtr* be assigned to the socket. Clients do not normally require that an address be assigned to a socket. However, servers usually require that the socket be bound to a “well known” address. The port number may be any port number between 0 and 65535. Multiple sockets cannot bind to the same port with different IP addresses (as might be allowed in UNIX)

Parameters

socketDescriptor	The socket descriptor to assign an IP address and port number to.
addressPtr	The pointer to the structure containing the address to assign.
addressLength	The length of the address structure.

Returns

0	Success
-1	An error occurred

bind can fail for any of the following reasons:

EADDRINUSE	The specified address is already in use.
EBADF	*socketDescriptor* is not a valid descriptor.
EINVAL	One of the passed parameters is invalid, or socket is already bound.
EINPROGRESS	bind is already running.

check_error

Get Error Code for Current Task

Syntax

```
#include <ctools.h>
UINT32 check_error(void);
```

Description

The **check_error** function returns the error code for the current task. The error code is set by various I/O routines, when errors occur. A separate error code is maintained for each task.

Notes

Some routines in the standard C library, return errors in the global variable **errno**. This variable is not unique to a task, and may be modified by another task, before it can be read.

See Also

checksum

Calculate a Checksum

Syntax

```
#include <ctools.h>
UINT16 checksum(UCHAR *start, UCHAR *end, UINT16 algorithm);
```

Description

The **checksum** function calculates a checksum on memory. The memory starts at the byte pointed to by *start*, and ends with the byte pointed to by *end*. The *algorithm* may be one of:

ADDITIVE	16 bit byte-wise sum
CRC_16	CRC-16 polynomial checksum
CRC_CCITT	CRC-CCITT polynomial checksum
BYTE_EOR	8 bit byte-wise exclusive OR

The CRC checksums use the **crc_reverse** function.

Example

This function displays two types of checksums.

```
#include <ctools.h>

void checksumExample(void)
{
       char str[] = "This is a test";
       UINT16 sum;

       /* Display additive checksum */
       sum = checksum(str, str+strlen(str), ADDITIVE);
       fprintf(com1,"Additive checksum: %u\r\n", sum);

       /* Display CRC-16 checksum */
       sum = checksum(str, str+strlen(str), CRC_16);
       fprintf(com1,"CRC-16 checksum: %u\r\n", sum);
}
```

checkSFTranslationTable

Test for Store and Forward Configuration Errors

Syntax

```
#include <ctools.h>
struct SFTranslationStatus checkSFTranslationTable(void);
```

Description

The **checkSFTranslationTable** function checks all entries in the address translation table for validity. It detects the following errors:

The function returns a *SFTranslationStatus* structure. Refer to the **Structures and Types** section for a description of the fields in the *SFTranslationStatus* structure. The *code* field of the structure is set to one of the following. If there is an error, the *index* field is set to the location of the translation that is not valid.

Result code	***Meaning***
SF_VALID	All translations are valid
SF_NO_TRANSLATION	The entry defines re-transmission of the same message on the same port
SF_PORT_OUT_OF_RANGE	One or both of the interfaces is not valid
SF_STATION_OUT_OF_RANGE	One or both of the stations is not valid
SF_ALREADY_DEFINED	The translation already exists in the table
SF_INVALID_FORWARDING_IP	The forwarding IP address is invalid.

Notes

The ***TeleBUS Protocols User Manual*** describes store and forward messaging mode.

See Also

clearSFTranslationTable

Example

See the example for the **setSFTranslationEx** function.

clearAllForcing

Clear All Forcing (TelePACE firmware only)

Syntax

```
#include <ctools.h>
void clearAllForcing(void);
```

Description

The **clearAllForcing** function removes all forcing conditions from all I/O database registers.

The IO_SYSTEM resource must be requested before calling this function.

See Also

setForceFlag**,** getForceFlag**,** overrideDbase

clear_errors

Clear Serial Port Error Counters

Syntax

```
#include <ctools.h>
void clear_errors(UCHAR port);
```

Description

The **clear_errors** function clears the serial port error counters for the serial port specified by *port*. If *port* is not a valid serial port the function has no effect.

The IO_SYSTEM resource must be requested before calling this function.

clear_protocol_status

Clear Protocol Counters

Syntax

```
#include <ctools.h>
void clear_protocol_status(UCHAR port);
```

Description

The **clear_protocol_status** function clears the error and message counters for the serial port specified by *port*. If *port* is not a valid serial port the function has no effect.

The IO_SYSTEM resource must be requested before calling this function.

clearRegAssignment

Clear Register Assignment (TelePACE firmware only)

Syntax

```
#include <ctools.h>
void clearRegAssignment(void);
```

Description

The **clearRegAssignment** function erases the current Register Assignment. Call this function first before using the **addRegAssignment** function to create a new Register Assignment.

To save the Register Assignment with the controller settings in flash memory so that it is loaded on controller reset, call **flashSettingsSave** as shown in the example for addRegAssignment.

The IO_SYSTEM resource must be requested before calling this function.

See Also

addRegAssignment

Example

See example for **addRegAssignment**.

clearSFTranslationTable

Clear Store and Forward Translation Configuration

Syntax

```
#include <ctools.h>
void clearSFTranslationTable(void);
```

Description

The **clearSFTranslationTable** function clears all entries in the store and forward translation table.

To save these settings with the controller settings in flash memory so that they are loaded on controller reset, call **flashSettingsSave** as shown below.

```
request_resource(FLASH_MEMORY);
flashSettingsSave(CS_RUN);
release_resource(FLASH_MEMORY);
```

Notes

The ***TeleBUS Protocols User Manual*** describes store and forward messaging mode.

The IO_SYSTEM resource must be requested before calling this function.

See Also

checkSFTranslationTable

clearStatusBit

Clear Bits in Controller Status Code

Syntax

```
#include <ctools.h>
UINT16 clearStatusBit(UINT16 bitMask);
```

Description

The **clearStatusBit** function clears the bits indicated by *bitMask* in the controller status code. When the status code is non-zero, the STAT LED blinks a binary sequence corresponding to the code. If *code* is zero, the STAT LED turns off.

The function returns the value of the status register.

Notes

The status output opens if *code* is non-zero. Refer to the ***System Hardware Manual*** for more information.

The binary sequence consists of short and long flashes of the error LED. A short flash of 1/10th of a second indicates a binary zero. A longer flash of approximately 1/2 of a second indicates a binary one. The least significant digit is output first. As few bits as possible are displayed – all leading zeros are ignored. There is a two-second delay between repetitions.

The STAT LED is located on the top left hand corner of the controller board.

Bits 0, 1 and 2 of the status code are used by the controller firmware. Attempting to control these bits will result in indeterminate operation.

See Also

setStatusBit**,** getStatusBit

clear_tx

Clear Serial Port Transmit Buffer

Syntax

```
#include <ctools.h>
void clear_tx(UCHAR port);
```

Description

The **clear_tx** function clears the transmit buffer for the serial port specified by *port*. If *port* is not a valid serial port the function has no effect.

close

Syntax

```
#include <ctools.h>
int close
(
      int socketDescriptor
);
```

Function Description

This function is used to close a socket.

Parameters

socketDescriptor	The socket descriptor to close

Returns

0	Operation completed successfully
-1	An error occurred

close can fail for the following reasons:

TM_EBADF	The socket descriptor is invalid.
TM_ESHUTDOWN	A write shutdown has already been performed on the socket (TCP socket only).
TM_EALREAY	A previous **close** call is already in progress.
TM_ECONNABORTED	The TCP connection was reset because the linger option was on with a timeout value of 0 (TCP socket only).
TM_ETIMEDOUT	The linger option was on with a non-zero timeout value, and the linger timeout expired before the TCP close handshake with the remote host could complete (blocking TCP socket only).

connect

Syntax

```
#include <ctools.h>
int connect
(
      int socketDescriptor,
      const struct sockaddr * addressPtr,
      int address_Length
);
```

Function Description

The parameter *socketDescriptor* is a socket. If it is of type SOCK_DGRAM, **connect** specifies the peer with which the socket is to be associated; this address is the address to which datagrams are to be sent if a receiver is not explicitly designated; **it is the only address from which datagrams are to be received**. If the socket *socketDescriptor* is of type SOCK_STREAM, **connect** attempts to make a connection to another socket (either local or remote). The other socket is specified by *addressPtr*. *addressPtr* is a pointer to the IP address and port number of the remote or local socket. If *socketDescriptor* is not bound, then it will be bound to an address selected by the underlying transport provider. Generally, stream sockets may successfully **connect** only once; datagram sockets may use **connect** multiple times to change their association. Datagram sockets may dissolve the association by connecting to a null address.

Note that a ***non –blocking*** connect is allowed. In this case, if the connection has not been established, the **connect** call will fail with a EINPROGRESS error code. Additional calls to **connect** will fail with EALREADY error code, as long as the connection has not completed. When the connection has completed, additional calls to **connect** will return with no error to indicate that the connection is now established.

Parameters

socketDescriptor	The socket descriptor to assign a name (port number) to.
addressPtr	The pointer to the structure containing the address to connect to for TCP. For UDP it is the default address to send to and the only address to receive from.
addressLength	The length of the address structure.

Returns

0 Success

-1 An error occurred.

connect can fail for any of the following reasons:

EADDRINUSE	The socket address is already in use. The calling program should close the socket descriptor, and issue another **socket** call to obtain a new descriptor before attempting another **connect** call.
EADDRNOTAVAIL	The specified address is not available on the remote / local machine.
EAFNOSUPPORT	Addresses in the specified address family cannot be used with this socket.

EINPROGRESS	The socket is non-blocking and the current connection attempt has not yet been completed.
EALREADY	The socket is non-blocking and a previous connection attempt has not yet been completed.
EBADF	*socketDescriptor* is not a valid descriptor.
ECONNREFUSED	The attempt to connect was forcefully rejected. The calling program should close the socket descriptor, and issue another socket call to obtain a new descriptor before attempting another **connect** call.
EPERM	Cannot call **connect** after **listen** call.
EINVAL	One of the parameters is invalid
EISCONN	The socket is already connected. The calling program should close the socket descriptor, and issue another socket call to obtain a new descriptor before attempting another **connect** call.
EHOSTUNREACH	No route to the host we want to connect to.
EPROTOTYPE	The socket referred to by *addressPtr* is a socket of a type other than type *socketDescriptor* (for example, *socketDescriptor* is a SOCK_DGRAM socket, while *addressPtr* refers to a SOCK_STREAM socket).
ETIMEDOUT	Connection establishment timed out, without establishing a connection. The calling program should close the socket descriptor, and issue another **socket** call to obtain a new descriptor before attempting another **connect** call.

crc_reverse

Calculate a CRC Checksum

Syntax

```
#include <ctools.h>
UINT16 crc_reverse(UCHAR *start, UCHAR *end, UINT16 poly, UINT16
   initial);
```

Description

The **crc_reverse** function calculates a CRC type checksum on memory using the reverse algorithm. The memory starts at the byte pointed to by *start*, and ends with the byte pointed to by *end*. The generator polynomial is specified by *poly*. *poly* may be any value, but must be carefully chosen to ensure good error detection. The checksum accumulator is set to *initial* before the calculation is started.

Notes

The reverse algorithm is named for the direction bits are shifted. In the reverse algorithm, bits are shifted towards the least significant bit. This produces different checksums than the classical, or forward algorithm, using the same polynomials.

See Also

checksum

create_task

Create a New Task

Syntax

```
#include <ctools.h>
INT32 create_task(void *function, UINT32 priority, UINT32 type, UINT32
   stack);
```

Description

The **create_task** function allocates stack space for a task and places the task on the ready queue. *function* specifies the start address of the routine to be executed. The task will execute immediately if its priority is lower than the current executing task.

priority is an execution priority between 0 and 254 for the created task. The lowest priority is 254, and the highest priority is 0. The 255 task priority levels aid in scheduling task execution. See the notes below for recommended priority values.

type specifies if the task is ended when an application program is stopped. Valid values for *type* are:

SYSTEM	System tasks do not terminate when the program stops.
`applicationGroup`	Application tasks terminate when the program stops. Use this global variable for all calls to create_task by the same application. The operating system assigns a unique value to `applicationGroup` when it is defined in appstart.cpp.

It is recommended that only application type tasks be created.

The *stack* parameter specifies how many stack blocks are allocated for the task. Each stack block is 512 bytes.

The **create_task** function returns the task ID (TID) of the task created. If an error occurs, -1 is returned.

Notes

Refer to the **Real Time Operating System** section for more information on tasks.

Note that the **main** task and the Ladder Logic and I/O scanning task have a priority of 100. If the created task is continuously running processing code, create the task with a priority of 100. The scheduling algorithm of the operating system will give each task of the same priority time slices to share the CPU.

For tasks such as a protocol handler, that wait for an event using the **wait_event** or **receive_message** function, a priority higher than 100 (e.g. 75) may be selected without blocking other lower priority tasks.

The number of stack blocks required depends on the functions called within the task, and the size of local variables created. Most tasks require 2 stack blocks. If the **fprintf** function is used, then at least 5 stack blocks are required. Add local variable usage to these limits, if large local arrays or structures are created. Large structures and arrays are usually best handled as static global variables within the task source file. (The variables are global to all functions in the task, but cannot be seen by functions in other files.)

Additional stack space may be made available by disabling unused protocol tasks. See the section **Program Development** or the **set_protocol** function for more information.

See Also

end_task

Example

See the Create Task Example in the Examples section.

databaseRead

Read Value from I/O Database

Syntax

```
#include <ctools.h>
BOOLEAN databaseRead(UINT16 addrMode, UINT16 address, INT16 *
value);
```

Description

The **databaseRead** function reads a value from the database. *addrMode* specifies the method of addressing the database. *address* specifies the location in the database. The table below shows the valid address modes and ranges

Type	Address Ranges	Register Size
MODBUS	00001 to NUMCOIL 10001 to 10000 + NUMSTATUS 30001 to 30000 + NUMINPUT 40001 to 40000 + NUMHOLDING	1 bit 1 bit 16 bit 16 bit
LINEAR	0 to NUMLINEAR-1	16 bit

Notes

The function **databaseRead** returns TRUE if the requested database value was read. FALSE is returned if the requested database entry could not be read. If the specified register is currently forced, **databaseRead** reads the forced register value into the memory pointed to by `value`.

The I/O database is not modified when the controller is reset. It is a permanent storage area, which is maintained during power outages.

The IO_SYSTEM resource must be requested before calling this function.

See Also

databaseWrite

databaseWrite

Write Value to I/O Database

Syntax

```
#include <ctools.h>
BOOLEAN databaseWrite(UINT16 addrMode, UINT16 address, INT16
value);
```

Description

The **databaseWrite** function writes a value to the database. *addrMode* specifies the method of addressing the database. *address* specifies the location in the database. The table below shows the valid address modes and ranges

Type	Address Ranges	Register Size
MODBUS	00001 to NUMCOIL 10001 to 10000 + NUMSTATUS 30001 to 30000 + NUMINPUT 40001 to 40000 + NUMHOLDING	1 bit 1 bit 16 bit 16 bit
LINEAR	0 to NUMLINEAR-1	16 bit

Notes

The function **databaseWrite** returns TRUE if the requested database value was written. FALSE is returned if the requested database entry could not be written.

The I/O database is not modified when the controller is reset. It is a permanent storage area, which is maintained during power outages.

The IO_SYSTEM resource must be requested before calling this function.

See Also

databaseRead

datalogCreate

Create Data Log Function

Syntax

```
#include <ctools.h>
DATALOG_STATUS datalogCreate(
      UINT16 logID,
      DATALOG_CONFIGURATION * pLogConfiguration);
```

Description

This function creates a data log with the specified configuration. The data log is created in the data log memory space.

The function has two parameters. *logID* specifies the data log to be created. The valid range is 0 to 15. *pLogConfiguration* points to a structure with the configuration for the data log.

The function returns the status of the operation.

Notes

The configuration of an existing data log cannot be changed. The log must be deleted and recreated to change the configuration.

All data logs are stored in memory from a pool for all data logs. If there is insufficient memory the creation operation fails. The function returns DLS_NOMEMORY.

If the data log already exists the creation operation fails. The function returns DLS_EXISTS.

If the log ID is not valid the creation operation fails. The function returns DLS_BADID.

If the configuration is not valid the creation operation fails. The function returns DLS_BADCONFIG.

See Also

datalogDelete, datalogSettings

Example

This program creates a data log and writes one record to it.

```
#include <ctools.h>

/* Structure used to copy one record into data log */
struct dataRecord
{
      UINT16 value1;
      INT32  value2;
      double value3;
      float  value4;
      float  value5;
};

int main(void)
{
```

```
        UINT16 logID;
        DATALOG_CONFIGURATION dLogConfig;/* log configuration */
        struct dataRecord data;          /* sample record */

        /* Assign a number to the data log */
        logID = 10;

        /* Fill in the log configuration structure */
        dLogConfig.records = 200;
        dLogConfig.fields = 5;
        dLogConfig.typesOfFields[0] = DLV_UINT16;
        dLogConfig.typesOfFields[1] = DLV_INT32;
        dLogConfig.typesOfFields[2] = DLV_DOUBLE;
        dLogConfig.typesOfFields[3] = DLV_FLOAT;
        dLogConfig.typesOfFields[4] = DLV_FLOAT;

        /* Assign some data for the log */
        data.value1 = 100;
        data.value2 = 200;
        data.value3 = 30000;
        data.value4 = 40;
        data.value5 = 50;

        if(datalogCreate(logID, &dLogConfig) == DLS_CREATED)
        {
           /* Start writing records in log */
           if(datalogWrite(logID, (UINT16 *)&data) )
           {
              /* one record was written in data log */
           }
        }
}
```

datalogDelete

Delete Data Log Function

Syntax

```
#include <ctools.h>
BOOLEAN datalogDelete(UINT16 logID);
```

Description

This function destroys the specified data log. The memory used by the data log is returned to the freed.

The function has one parameter. *logID* specifies the data log to be deleted. The valid range is 0 to 15.

The function returns TRUE if the data log was deleted. The function returns FALSE if the log ID is not valid or if the log had not been created.

See Also

Example

This program shows the only way to change the configuration of an existing log, which is to delete the log and recreate the data log.

```
#include <ctools.h>

int main(void)
{
       UINT16 logID;
       DATALOG_CONFIGURATION dLogConfig;

       /* Select logID #10 */
       logID = 10;

       /* Read the configuration of logID #10 */
       if(datalogSettings(logID, &dLogConfig))
       {
          if(dLogConfig.typesOfFields[0] == DLV_INT16)
          {
             /* Wrong type. Delete log and create new one */
             if(datalogDelete(logID) )
             {
                /* Re-enter the log configuration */
                dLogConfig.records = 200;
                dLogConfig.fields = 5;
                dLogConfig.typesOfFields[0] = DLV_UINT16;
                dLogConfig.typesOfFields[1] = DLV_INT32;
                dLogConfig.typesOfFields[2] = DLV_DOUBLE;
                dLogConfig.typesOfFields[3] = DLV_FLOAT;
                dLogConfig.typesOfFields[4] = DLV_FLOAT;
                datalogCreate(logID, &dLogConfig);
             }
             else
             {
                /* could not delete log */
```

```
            }
        }
    }
    else
    {
        /* Could not read settings */
    }
}
```

datalogPurge

Purge Data Log Function

Syntax

```
#include <ctools.h>
BOOLEAN datalogPurge(
       UINT16 logID,
       BOOLEAN purgeAll,
       UINT32 sequenceNumber);
```

Description

This function removes records from a data log. The function can remove all the records, or a group of records starting with the oldest in the log.

The function has three parameters. *logID* specifies the data log. The valid range is 0 to 15. If *purgeAll* is TRUE, all records are removed, otherwise the oldest records are removed. *sequenceNumber* specifies the sequence number of the most recent record to remove. All records up to and including this record are removed. This parameter is ignored if *purgeAll* is TRUE.

The function returns TRUE if the operation succeeds. The function returns FALSE if the log ID is invalid, if the log has not been created, or if the sequence number cannot be found in the log.

Notes

Purging the oldest records in the log is usually done after reading the log. The sequence number used is that of the last record read from the log. This removes the records that have been read and leaves any records added since the records were read.

If the sequence number specifies a record that is not in the log, no records are removed.

See Also

datalogReadStart, datalogReadNext, datalogWrite

Example

```
#include <ctools.h>
int main(void)
{
       UINT16 logID;
       UINT32 sequenceNumber;
       BOOLEAN purgeAll;

       /* select data log to be purged */
       logID = 10;

       /* set flag to purge only part of data log */
       purgeAll = FALSE;

       /* purge the oldest 150 records */
       sequenceNumber = 150;

       if(datalogPurge(logID, purgeAll, sequenceNumber))
       {
```

```
            /* Successful at purging the first 150 records of log. */
            /* Start writing records again. */
        }

        /* To purge the entire data log, set flag to TRUE */
        purgeAll = TRUE;

        /* call function with same parameters */
        if( datalogPurge(logID, purgeAll, sequenceNumber) )
        {
            /* Successful at purging the entire data log. */
            /* Start writing records again. */
        }
}
```

datalogReadNext

Read Data Log Next Function

This function returns the next record in the data log.

Syntax

```
#include <ctools.h>
BOOLEAN datalogReadNext(
      UINT16 logID,
      UINT32 sequenceNumber,
      UINT32 * pSequenceNumber,
      UINT32 * pNextSequenceNumber,
      UINT16 * pData);
```

Description

This function reads the next record from the data log starting at the specified sequence number. The function returns the record with the specified sequence number if it is present in the log. If the record no longer exists it returns the next record in the log.

The function has five parameters. *logID* specifies the data log. The valid range is 0 to 15. *sequenceNumber* is sequence number of the record to be read. *pSequenceNumber* is a pointer to a variable to hold the sequence number of the record read. *pNextSequenceNumber* is a pointer to a variable to hold the sequence number of the next record in the log. This is normally used for the next call to this function. *pData* is a pointer to memory to hold the data read from the log.

The function returns TRUE if a record is read from the log. The function returns FALSE if the log ID is not valid, if the log has not been created or if there are no more records in the log.

Notes

Use the **datalogReadStart** function to obtain the sequence number of the oldest record in the data log.

The *pData* parameter must point to memory of sufficient size to hold all the data in a record.

It is normally necessary to call this function until it returns FALSE in order to read all the data from the log. This accommodates cases where data is added to the log while it is being read.

If data is read from the log at a slower rate than it is logged, it is possible that the sequence numbers of the records read will not be sequential. This indicates that records were overwritten between calls to read data.

The sequence number rolls over after reaching its maximum value.

See Also

datalogReadStart, datalogPurge, datalogWrite

Example

See the example for datalogReadStart.

datalogReadStart

Read Data Log Start Function

Syntax

```
#include <ctools.h>
BOOLEAN datalogReadStart(
      UINT16 logID,
      UINT32 * pSequenceNumber);
```

Description

This function returns the sequence number of the record at the start of the data log. This is the oldest record in the log.

The function has two parameters. *logID* specifies the data log. The valid range is 0 to 15. *pSequenceNumber* is a pointer to a variable to hold the sequence number.

The function returns TRUE if the operation succeeded. The function returns FALSE if the log ID is not valid or if the log has not been created.

Notes

Use the **datalogReadNext** function to read records from the log.

The function will return a sequence number even if the log is empty. In this case the next call to **datalogReadNext** will return no data.

See Also

datalogReadNext, datalogPurge, datalogWrite

Example

```
#include <ctools.h>
#include <stdlib.h>

int main(void)
{
      UINT16 logID, recordSize, *pData;
      UINT32 sequenceNumber, seqNumRead, nextSeqNum;

      /* Select data log #10 */
      logID = 10;

      /* Find first record in data log #10 and store
         its sequence number in sequenceNumber      */
      if(datalogReadStart(logID, &sequenceNumber))
      {
         /* Get the size of this record */
         if(datalogRecordSize(logID, &recordSize))
         {
            /* allocate memory of size recordSize */
            pData = (UINT16 *)malloc(recordSize);

            /* read this record */
            if(datalogReadNext(logID, sequenceNumber, &seqNumRead,
         &nextSeqNum, pData))
            {
```

```
                    /* use pData to access record contents */
                }
            }
        }
    }
```

datalogRecordSize

Data Log Record Size Function

Syntax

```
#include <ctools.h>
BOOLEAN datalogRecordSize(
      UINT16 logID,
      UINT16 * pRecordSize);
```

Description

This function returns the size of a record for the specified data log. The log must have been previously created with the **datalogCreate** function.

The function has two parameters. *logID* specifies the data log. The valid range is 0 to 15. *pRecordSize* points to a variable that will hold the size in bytes of each record in the log.

The function returns TRUE if the operation succeeded. The function returns FALSE if the log ID is invalid or if the data log does not exist.

Notes

This function is useful in determining how much memory must be allocated for a call to **datalogReadNext** or **datalogWrite**.

See Also

datalogSettings

Example

See the example for datalogReadStart.

datalogSettings

Data Log Settings Function

Syntax

```
#include <ctools.h>
BOOLEAN datalogSettings(
      UINT16 logID,
      DATALOG_CONFIGURATION * pLogConfiguration);
```

Description

This function reads the configuration of the specified data log. The log must have been previously created with the **datalogCreate** function.

The function has two parameters. *logID* specifies the data log. The valid range is 0 to 15. *pLogConfiguration* points to a structure that will hold the data log configuration.

The function returns TRUE if the operation succeeded. The function returns FALSE if the log ID is invalid or if the data log does not exist.

Notes

The configuration of an existing data log cannot be changed. The log must be deleted and recreated to change the configuration.

See Also

datalogRecordSize

Example

See example for datalogDelete.

datalogWrite

Write Data Log Function

Syntax

```
#include <ctools.h>
BOOLEAN datalogWrite(
      UINT16 logID,
      UINT16 * pData);
```

Description

This function writes a record to the specified data log. The log must have been previously created with the **datalogCreate** function.

The function has two parameters. *logID* specifies the data log. The valid range is 0 to 15. *pData* is a pointer to the data to be written to the log. The amount of data copied using the pointer is determined by the configuration of the data log.

The function returns TRUE if the data is added to the log. The function returns FALSE if the log ID is not valid or if the log does not exist.

Notes

Refer to the **datalogCreate** function for details on the configuration of the data log.

If the data log is full, then the oldest record in the log is replaced with this record.

See Also

datalogReadStart, datalogReadNext, datalogPurge

Example

See the example for XXX.

dbase

Read Value from I/O Database

Syntax

```
#include <ctools.h>
INT16 dbase(UINT16 type, UINT16 address);
```

Description

The **dbase** function reads a value from the database. *type* specifies the method of addressing the database. *address* specifies the location in the database. The table below shows the valid address types and ranges

Type	Address Ranges	Register Size
MODBUS	00001 to NUMCOIL 10001 to 10000 + NUMSTATUS 30001 to 30000 + NUMINPUT 40001 to 40000 + NUMHOLDING	1 bit 1 bit 16 bit 16 bit
LINEAR	0 to NUMLINEAR-1	16 bit

Notes

If the specified register is currently forced, **dbase** returns the forced value for the register.

The I/O database is not modified when the controller is reset. It is a permanent storage area, which is maintained during power outages.

The IO_SYSTEM resource must be requested before calling this function.

See Also

setdbase

Example

```
#include <ctools.h>
int main(void)
{
      int a;
      request_resource(IO_SYSTEM);

      /* Read Modbus status input point */
      a = dbase(MODBUS, 10001);

      /* Read 16 bit register */
      a = dbase(LINEAR, 3020);

      /* Read 16 bit register beginning at first
      status register */
      a = dbase(LINEAR, START_STATUS);

      /* Read 6th input register */
      a = dbase(LINEAR, START_INPUT + 5);

      release_resource(IO_SYSTEM);
}
```

Dbase Handler Function

User Defined Dbase Handler Function

The dbase handler function is a user-defined function that handles reading of Modbus addresses not assigned in the ISaGRAF Dictionary. The function can have any name; *dbaseHandler* is used in the description below.

Syntax

```
#include <ctools.h>
BOOLEAN dbaseHandler(
   UINT16 address,
   INT * value
   )
```

Description

This function is called by the **dbase** function when one of the following conditions apply:

- There is no ISaGRAF application downloaded, or
- There is no ISaGRAF variable assigned to the specified Modbus address.

The function has two parameters:

- The *address* parameter is the Modbus address to be read.
- The *value* parameter is a pointer to an integer containing the current value at *address*.

If the address is to be handled, the handler function must return TRUE and the value pointed to by *value* must be set to the current value for the specified Modbus *address*.

If the address is not to be handled, the function must return FALSE and the value pointed to by *value* must be left unchanged.

Notes

The IO_SYSTEM resource must be requested before calling dbase, which calls this handler. Requesting the IO_SYSTEM resource ensures that only one task may call the handler at a time. Therefore, the function does not have to be re-entrant.

An array may be defined to store the current values for all Modbus addresses handled by this function. See the section *Data Storage* if a non-initialized data array is required.

See Also

installDbaseHandler

deallocate_envelope

Return Envelope to the RTOS

Syntax

```
#include <ctools.h>
void deallocate_envelope(envelope *penv);
```

Description

The **deallocate_envelope** function returns the envelope pointed to by *penv* to the pool of free envelopes maintained by the operating system.

See Also

allocate_envelope

Example

See the example for the **allocate_envelope** function.

dnpClearEventLogs

Clear DNP Event Log

Syntax

```
#include <ctools.h>
BOOLEAN dnpClearEventLogs(void);
```

Description

The **dnpClearEventLogs** function deletes all change events from the DNP change event buffers, for all point types.

dnpConnectionEvent

Report a DNP connection event

Syntax

```
#include <ctools.h>
void dnpConnectionEvent(
      UINT16 dnpAddress,
      DNP_CONNECTION_EVENT event);
```

Description

The **dnpConnectionEvent** function is used to report a change in connection status to DNP. This function is only used if a custom DNP connection handler has been installed.

`dnpAddress` is the address of the remote DNP station.

`event` is current connection status. The valid connection status settings are `DNP_CONNECTED`, and `DNP_DISCONNECTED`.

See Also

dnpInstallConnectionHandler

Example

See the `dnpInstallConnectionHandler` example.

dnpCreateAddressMappingTable

Create DNP Address Mapping Table

Syntax

```
#include <ctools.h>
      BOOLEAN dnpCreateAddressMappingTable (
        UINT16 size,
        CHAR enableMapChangeEvents);
```

Description

The **dnpCreateAddressMappingTable** function destroys any existing DNP address mapping table, and allocates memory for a new address mapping table according to the ‘size’ parameter.

Notes

DNP must be enabled before calling this function in order to create the DNP configuration.

The function returns TRUE if successful, FALSE otherwise.

dnpCreateMasterPollTable

Create DNP Master Poll Table

Syntax

```
#include <ctools.h>
      BOOLEAN dnpCreateMasterPollTable (
        UINT16 size);
```

Description

This function destroys any existing DNP master poll table, and allocates memory for a new table according to the ‘size’ parameter. The poll interval is set (in seconds).

Notes

DNP must be enabled before calling this function in order to create the DNP configuration.

The function returns TRUE if successful, FALSE otherwise.

dnpCreateRoutingTable

Create Routing Table

Syntax

```
#include <ctools.h>
BOOLEAN dnpCreateRoutingTable(
        UINT16 size);
```

Description

This function destroys any existing DNP routing table, and allocates memory for a new routing table according to the ‘size’ parameter.

Notes

DNP must be enabled before calling this function in order to create the DNP configuration.

The function returns TRUE if successful, FALSE otherwise.

Example

See the example in the ***dnpGetConfiguration*** section.

dnpGenerateChangeEvent

Generate DNP Change Event

Syntax

```
BOOLEAN dnpGenerateChangeEvent(
      DNP_POINT_TYPE pointType,
      UINT16 pointAddress
);
```

Description

The `dnpGenerateChangeEvent` function generates a change event for the DNP point specified by `pointType` and `pointAddress`.

`pointType` specifies the type of DNP point. Allowed values are:

BI_POINT	binary input
AI16_POINT	16 bit analog input
AI32_POINT	32 bit analog input
AISF_POINT	short float analog input
CI16_POINT	16 bit counter output
CI32_POINT	32 bit counter output

`pointAddress` specifies the DNP address of the point.

A change event is generated for the specified point (with the current time and current value), and stored in the DNP event buffer.

The format of the event will depend on the Event Reporting Method and Class of Event Object that have been configured for the point.

The function returns TRUE if the event was generated. It returns FALSE if the DNP point is invalid, or if the DNP configuration has not been created.

Notes

DNP must be enabled before calling this function in order to create the DNP configuration.

dnpGenerateEventLog

Generates a change event for the DNP point

Syntax

```
#include <ctools.h>
      BOOLEAN dnpGenerateEventLog(
              UINT16 pointType,
              UINT16 pointAddress);
```

Description

The **dnpGenerateEventLog** function generates a change event for the DNP point.

Notes

Example

See example in the dnpGetConfiguration function section.

dnpGetAI16Config

Get DNP 16-bit Analog Input Configuration

Syntax

```
#include <ctools.h>
BOOLEAN dnpGetAI16Config(
        UINT16 point,
        dnpAnalogInput * pAnalogInput);
```

Description

The **dnpGetAI16Config** function reads the configuration of a DNP 16-bit analog input point.

The function has two parameters: the point number; and a pointer to an analog input point configuration structure.

The function returns TRUE if the configuration was read. It returns FALSE if the point number is not valid, if the pointer is NULL, or if DNP configuration has not been created.

Notes

DNP must be enabled before calling this function in order to create the DNP configuration.

Example

See example in the dnpGetConfiguration function section.

dnpGetAI32Config

Get DNP 32-bit Analog Input Configuration

Syntax

```
#include <ctools.h>
BOOLEAN dnpGetAI32Config(
        UINT32 point,
        dnpAnalogInput * pAnalogInput);
```

Description

The **dnpGetAI32Config** function reads the configuration of a DNP 32-bit analog input point.

The function has two parameters: the point number; and a pointer to an analog input point configuration structure.

The function returns TRUE if the configuration was read. It returns FALSE if the point number is not valid, if the pointer is NULL, or if DNP configuration has not been created.

Notes

DNP must be enabled before calling this function in order to create the DNP configuration.

See Also

dnpSaveAI32Config

Example

See example in the dnpGetConfiguration function section.

dnpGetAISFConfig

Get Short Floating Point Analog Input Configuration

Syntax

```
#include <ctools.h>
BOOLEAN dnpGetAISFConfig (
        UINT16 point,
        dnpAnalogInput *pAnalogInput);
```

Description

The **dnpGetAISFConfig** function reads the configuration of a DNP short floating point analog input point.

The function has two parameters: the point number, and a pointer to a configuration structure.

The function returns TRUE if the configuration was successfully read, or FALSE otherwise (if the point number is not valid, or pointer is NULL, or if the DNP configuration has not been created).

Notes

DNP must be enabled before calling this function in order to create the DNP configuration.

dnpGetAO16Config

Get DNP 16-bit Analog Output Configuration

Syntax

```
#include <ctools.h>
BOOLEAN dnpGetAO16Config(
        UINT16 point,
        dnpAnalogOutput * pAnalogOutput);
```

Description

The dnpGetAO16Config function reads the configuration of a DNP 16-bit analog output point.

The function has two parameters: the point number; and a pointer to an analog output point configuration structure.

The function returns TRUE if the configuration was read. It returns FALSE if the point number is not valid, if the pointer is NULL, or if DNP configuration has not been created.

Notes

DNP must be enabled before calling this function in order to create the DNP configuration.

See Also

dnpSaveAO16Config

Example

See example in the dnpGetConfiguration function section.

dnpGetAO32Config

Get DNP 32-bit Analog Output Configuration

Syntax

```
#include <ctools.h>
BOOLEAN dnpGetAO32Config(
        UINT32 point,
        dnpAnalogOutput * pAnalogOutput);
```

Description

The **dnpGetAO32Config** function reads the configuration of a DNP 32-bit analog output point.

The function has two parameters: the point number; and a pointer to an analog output point configuration structure.

The function returns TRUE if the configuration was read. It returns FALSE if the point number is not valid, if the pointer is NULL, or if DNP configuration has not been created.

Notes

DNP must be enabled before calling this function in order to create the DNP configuration.

See Also

dnpSaveAO32Config

Example

See example in the dnpGetConfiguration function section.

dnpGetAOSFConfig

Get Short Floating Point Analog Output Configuration

Syntax

```
#include <ctools.h>
BOOLEAN dnpGetAOSFConfig (
        UINT16 point,
        dnpAnalogOutput *pAnalogOutput);
```

Description

The **dnpGetAOSFConfig** function reads the configuration of a DNP short floating point analog output point.

The function has two parameters: the point number, and a pointer to a configuration structure.

The function returns TRUE if the configuration was successfully read, or FALSE otherwise (if the point number is not valid, or pointer is NULL, or if the DNP configuration has not been created).

Notes

DNP must be enabled before calling this function in order to create the DNP configuration.

dnpGetBIConfig

Get DNP Binary Input Configuration

Syntax

```
#include <ctools.h>
BOOLEAN dnpGetBIConfig(
        UINT16 point,
        dnpBinaryInput * pBinaryInput);
```

Description

The **dnpGetBIConfig** function reads the configuration of a DNP binary input point.

The function has two parameters: the point number; and a pointer to a binary input point configuration structure.

The function returns TRUE if the configuration was read. It returns FALSE if the point number is not valid, if the pointer is NULL, or if DNP configuration has not been created.

Notes

DNP must be enabled before calling this function in order to create the DNP configuration.

See Also

dnpSaveBIConfig

Example

See example in the dnpGetConfiguration function section.

dnpGetBIConfigEx

Read DNP Binary Input Extended Point

Syntax

```
BOOLEAN dnpGetBIConfigEx(
      UINT16 point,
      dnpBinaryInputEx *pBinaryInput
);
```

Description

This function reads the configuration of an extended DNP Binary Input point.

The function has two parameters: the point number, and a pointer to an extended binary input point configuration structure.

The function returns TRUE if the configuration was successfully read. It returns FALSE if the point number is not valid, if the configuration is not valid, or if the DNP configuration has not been created.

This function supersedes `dnpGetBIConfig`.

dnpGetBOConfig

Get DNP Binary Output Configuration

Syntax

```
#include <ctools.h>
BOOLEAN dnpGetBOConfig(
        UINT16 point,
        dnpBinaryOutput * pBinaryOutput);
```

Description

The **dnpGetBOConfig** function reads the configuration of a DNP binary output point.

The function has two parameters: the point number; and a pointer to a binary output point configuration structure.

The function returns TRUE if the configuration was read. It returns FALSE if the point number is not valid, if the pointer is NULL, or if DNP configuration has not been created.

Notes

DNP must be enabled before calling this function in order to create the DNP configuration.

See Also

dnpSaveBOConfig

Example

See example in the dnpGetConfiguration function section.

dnpGetCI16Config

Get DNP 16-bit Counter Input Configuration

Syntax

```
#include <ctools.h>
BOOLEAN dnpGetCI16Config(
        UINT16 point,
        dnpCounterInput * pCounterInput);
```

Description

The **dnpGetCI16Config** function reads the configuration of a DNP 16-bit counter input point.

The function has two parameters: the point number; and a pointer to a counter input point configuration structure.

The function returns TRUE if the configuration was read. It returns FALSE if the point number is not valid, if the pointer is NULL, or if DNP configuration has not been created.

Notes

DNP must be enabled before calling this function in order to create the DNP configuration.

See Also

dnpSaveCI16Config

Example

See example in the dnpGetConfiguration function section.

dnpGetCI32Config

Get DNP 32-bit Counter Input Configuration

Syntax

```
#include <ctools.h>
BOOLEAN dnpGetCI32Config(
        UINT32 point,
        dnpCounterInput * pCounterInput);
```

Description

The **dnpGetCI32Config** function reads the configuration of a DNP 32-bit counter input point.

The function has two parameters: the point number; and a pointer to a counter input point configuration structure.

The function returns TRUE if the configuration was read. It returns FALSE if the point number is not valid, if the pointer is NULL, or if DNP configuration has not been created.

Notes

DNP must be enabled before calling this function in order to create the DNP configuration.

See Also

dnpSaveCI32Config

Example

See example in the dnpGetConfiguration function section.

dnpGetConfiguration

Get DNP Configuration

Syntax

```
#include <ctools.h>
BOOLEAN dnpGetConfiguration(
        dnpConfiguration * pConfiguration);
```

Description

The **dnpGetConfiguration** function reads the DNP configuration.

The function has one parameter: a pointer to a DNP configuration structure.

The function returns TRUE if the configuration was read and FALSE if an error occurred.

See Also

dnpSaveConfiguration

Example

The following program demonstrates how to configure DNP for operation on com2. To illustrate creation of points it uses a sequential mapping of Modbus registers to points. This is not required. Any mapping may be used.

```
int main(void)
{
       UINT16 index;                        /* loop index */
       struct prot_settings settings;       /* protocol settings */
       dnpConfiguration configuration;      /* configuration settings */
       dnpBinaryInput binaryInput;          /* binary input settings */
       dnpBinaryOutput binaryOutput;        /* binary output settings */
       dnpAnalogInput analogInput;          /* analog input settings */
       dnpAnalogOutput analogOutput;        /* analog output settings */
       dnpCounterInput counterInput;        /* counter input settings */

       /* Stop any protocol currently active on com port 2 */
       get_protocol(com2,&settings);
       settings.type = NO_PROTOCOL;
       set_protocol(com2,&settings);

       /* Load the Configuration Parameters */
       configuration.masterAddress         = DEFAULT_DNP_MASTER;
       configuration.rtuAddress            = DEFAULT_DNP_RTU;
       configuration.datalinkConfirm       = TRUE;
       configuration.datalinkRetries       = DEFAULT_DLINK_RETRIES;
       configuration.datalinkTimeout       = DEFAULT_DLINK_TIMEOUT;

       configuration.operateTimeout        = DEFAULT_OPERATE_TIMEOUT;
       configuration.applicationConfirm    = TRUE;
       configuration.maximumResponse       = DEFAULT_MAX_RESP_LENGTH;
       configuration.applicationRetries    = DEFAULT_APPL_RETRIES;
       configuration.applicationTimeout    = DEFAULT_APPL_TIMEOUT;
       configuration.timeSynchronization   = TIME_SYNC;

       configuration.BI_number             = 8;
       configuration.BI_cosBufferSize      = DEFAULT_COS_BUFF;
       configuration.BI_soeBufferSize      = DEFAULT_SOE_BUFF;
       configuration.BO_number             = 8;
       configuration.CI16_number           = 24;
       configuration.CI16_bufferSize       = 48;
```

```
configuration.CI32_number           = 12;
configuration.CI32_bufferSize       = 24;
configuration.AI16_number           = 24;
configuration.AI16_reportingMethod  = CURRENT_VALUE;
configuration.AI16_bufferSize       = 24;
configuration.AI32_number           = 12;
configuration.AI32_reportingMethod  = CURRENT_VALUE;
configuration.AI32_bufferSize       = 12;
configuration.AO16_number           = 8;
configuration.AO32_number           = 8;

configuration.unsolicited           = TRUE;

configuration.holdTime              = DEFAULT_HOLD_TIME;
configuration.holdCount             = DEFAULT_HOLD_COUNT;

dnpSaveConfiguration(&configuration);

/* Start DNP protocol on com port 2 */
get_protocol(com2,&settings);
settings.type = DNP;
set_protocol(com2,&settings);

/* Save port settings so DNP protocol will automatically start */
request_resource(IO_SYSTEM);
save(EEPROM_RUN);
release_resource(IO_SYSTEM);

/* Configure Binary Output Points */
for (index = 0; index < configuration.BO_number; index++)
{
       binaryOutput.modbusAddress1 = 1 + index;
       binaryOutput.modbusAddress2 = 1 + index;
       binaryOutput.controlType    = NOT_PAIRED;

       dnpSaveBOConfig(index, &binaryOutput);
}

/* Configure Binary Input Points */
for (index = 0;index < configuration.BI_number; index++)
{
       binaryInput.modbusAddress = 10001 + index;
       binaryInput.class         = CLASS_1;
       binaryInput.eventType     = COS;

       dnpSaveBIConfig(index, &binaryInput);
}

/* Configure 16 Bit Analog Input Points */
for (index = 0; index < configuration.AI16_number; index++)
{
       analogInput.modbusAddress  = 30001 + index;
       analogInput.class          = CLASS_2;
       analogInput.deadband       = 1;

       dnpSaveAI16Config(index, &analogInput);
}

/* Configure32 Bit Analog Input Points */
for (index = 0; index < configuration.AI32_number; index++)
{
       analogInput.modbusAddress  = 30001 + index * 2;
       analogInput.class          = CLASS_2;
       analogInput.deadband       = 1;

       dnpSaveAI32Config(index,&analogInput);
}

/* Configure 16 Bit Analog Output Points */
for (index = 0;index < configuration.AO16_number; index++)
{
```

```
            analogOutput.modbusAddress = 40001 + index;

            dnpSaveAO16Config(index, &analogOutput);
        }

        /* Configure 32 Bit Analog Output Points */
        for (index = 0; index < configuration.AO32_number; index++)
        {
            analogOutput.modbusAddress = 40101 + index * 2;

            dnpSaveAO32Config(index, &analogOutput);
        }

        /* Configure 16 Bit Counter Input Points */
        for (index = 0; index < configuration.CI16_number; index++)
        {
            counterInput.modbusAddress = 30001 + index;
            counterInput.class         = CLASS_3;
            counterInput.threshold     = 1;

            dnpSaveCI16Config(index, &counterInput);
        }

        /* Configure 32 bit Counter Input Points */
        for (index = 0; index < configuration.CI32_number; index++)
        {
            counterInput.modbusAddress = 30001 + index * 2;
            counterInput.class         = CLASS_3;
            counterInput.threshold     = 1;

            dnpSaveCI32Config(index, &counterInput);
        }

        /* add additional initialization code for your application here ... */

        /* loop forever */
        while (TRUE)
        {
            /* add additional code for your application here ... */

            /* allow other tasks of this priority to execute */
            release_processor();
        }
        return;
}
```

dnpGetConfigurationEx

Read DNP Extended Configuration

Syntax

```
BOOLEAN dnpGetConfigurationEx (
      dnpConfigurationEx *pDnpConfigurationEx
);
```

Description

This function reads the extended DNP configuration parameters.

The function has one parameter: a pointer to the DNP extended configuration structure.

The function returns TRUE if the configuration was successfully read, or FALSE otherwise (if the pointer is NULL, or if the DNP configuration has not been created).

Notes

DNP must be enabled before calling this function in order to create the DNP configuration.

This function supersedes the `dnpGetConfiguration` function.

dnpGetRuntimeStatus

Get DNP Runtime Status

Syntax

```
#include <ctools.h>
BOOLEAN dnpGetRuntimeStatus(
  DNP_RUNTIME_STATUS *status);
```

Description

The **dnpGetRuntimeStatus** function reads the current status of all DNP change event buffers, and returns information in the status structure.

DNP must be enabled before calling this function in order to create the DNP configuration.

Example

See the example in the ***dnpGetConfiguration*** section

dnpInstallConnectionHandler

Configures the connection handler for DNP

Syntax

```
#include <ctools.h>
void dnpInstallConnectionHandler(
          void (* handler)(
                UINT16 dnpAddress,
                DNP_CONNECTION_EVENT event));
```

Description

This function installs a handler that will permit user-defined actions to occur when DNP requires a connection, message confirmation is received, or a timeout occurs.

`handler` is a pointer to the handler function. If function is NULL the handler is disabled.

The function has no return value.

Notes

The handler function must process the event and return immediately. If the required action involves waiting this must be done outside of the handler function. See the example below for one possible implementation.

The application must disable the handler when the application ends. This prevents the protocol driver from calling the handler while the application is stopped. Call the `dnpInstallConnectionHandler` with a NULL pointer. The usual method is to create a task exit handler function to do this. See the example below for details.

The handler function has one parameter.

- `event` is DNP event that has occurred. It may be one of `DNP_CONNECTION_REQUIRED`, `DNP_MESSAGE_COMPLETE`, or `DNP_MESSAGE_TIMEOUT`. See the structure definition for the meaning of these events.

The handler function has no return value.

By default no connection handler is installed and no special steps are taken when DNP requires a connection, receives a message confirmation, or a timeout occurs.

See Also

dnpConnectionEvent

Example

This example shows how a C application can handle the events and inform a logic application of the events. The logic application is responsible for making and ending the dial-up connection.

The program uses the following registers.

- 10001 turns on when a connection is requested by DNP for unsolicited reporting.
- 10002 turns on when the unsolicited report is complete.

- 10003 turns on when the unsolicited report is fails.
- The ladder logic program turns on register 1 when the connection is complete and turns off the register when the connection is broken.

```
/* ---------------------------------------------------------------------
   dnp.c
   Demonstration program for using the DNP connection handler.

   Copyright 2001, Control Microsystems Inc.
   --------------------------------------------------------------------- */

/* ---------------------------------------------------------------------
   Include Files
   --------------------------------------------------------------------- */
#include <ctools.h>

/* ---------------------------------------------------------------------
   Constants
   --------------------------------------------------------------------- */
#define CONNECTION_REQUIRED  10001    /* register for signaling connection
                required */
#define MESSAGE_COMPLETE     10002    /* register for signaling unsolicited
                message is complete */
#define MESSAGE_FAILED       10003    /* register for signaling unsolicited
                message failed */
#define CONNECTION_STATUS    1        /* connection status register */

/* ---------------------------------------------------------------------
   Private Functions
   --------------------------------------------------------------------- */

/* ---------------------------------------------------------------------
   sampleDNPHandler

   This function is the user defined DNP connection handler.  It will be
   called by internal DNP routines when a connection is required, when
   confirmation of a message is received, and when a communication timeout
   occurs.

   The function takes a variable of type DNP_CONNECTION_EVENT as an input.
   This input instructs the handler as to what functionality is required.
   The valid choices are connection required (DNP_CONNECTION_REQUIRED),
   message confirmation received (DNP_MESSAGE_COMPLETE), and timeout occurred
   (DNP_MESSAGE_TIMEOUT).

   The function does not return any values.
   --------------------------------------------------------------------- */
static void sampleDNPHandler(DNP_CONNECTION_EVENT event)
{
       /* Determine what connection event is required or just occurred */
       switch(event)
       {
              case DNP_CONNECTION_REQUIRED:
                     /* indicate connection is needed and clear other bits */
                     request_resource(IO_SYSTEM);
                     setdbase(MODBUS, CONNECTION_REQUIRED, 1);
                     setdbase(MODBUS, MESSAGE_COMPLETE, 0);
                     setdbase(MODBUS, MESSAGE_FAILED, 0);
                     release_resource(IO_SYSTEM);
                     break;

              case DNP_MESSAGE_COMPLETE:
                     /* indicate message sent and clear other bits */
                     request_resource(IO_SYSTEM);
                     setdbase(MODBUS, CONNECTION_REQUIRED, 0);
                     setdbase(MODBUS, MESSAGE_COMPLETE, 1);
                     setdbase(MODBUS, MESSAGE_FAILED, 0);
                     release_resource(IO_SYSTEM);
                     break;
```

```
                case DNP_MESSAGE_TIMEOUT:
                        /* indicate message failed and clear other bits */
                        request_resource(IO_SYSTEM);
                        setdbase(MODBUS, CONNECTION_REQUIRED, 0);
                        setdbase(MODBUS, MESSAGE_COMPLETE, 0);
                        setdbase(MODBUS, MESSAGE_FAILED, 1);
                        release_resource(IO_SYSTEM);
                        break;

                default:
                        /* ignore invalid requests */
                        break;
        }
}

/* ------------------------------------------------------------------------
   Public Functions
   ------------------------------------------------------------------------ */

/* ------------------------------------------------------------------------
   main

   This function is the main task of a user application. It monitors a
   register from the ladder logic application. When the register value
   changes, the function signals DNP events.

   The function has no parameters.

   The function does not return.
   ------------------------------------------------------------------------ */
int main(void)
{
        int lastConnectionState;      /* last state of connection register */
        int currentConnectionState;   /* current state of connection register */

        /* install DNP connection handler */
        dnpInstallConnectionHandler(sampleDNPHandler);

        /* get the current connection state */
        lastConnectionState = dbase(MODBUS, CONNECTION_STATUS);

        /* loop forever */
        while (TRUE)
        {
                request_resource(IO_SYSTEM);

                /* get the current connection state */
                currentConnectionState = dbase(MODBUS, CONNECTION_STATUS);

                /* if the state has changed */
                if (currentConnectionState != lastConnectionState)
                {
                        /* if the connection is active */
                        if (currentConnectionState)
                        {
                                /* Inform DNP that a connection exists */
                                dnpConnectionEvent(DNP_CONNECTED);

                                /* clear the request flag */
                                setdbase(MODBUS, CONNECTION_REQUIRED, 0);
                        }
                        else
                        {
                                /* Inform DNP that the connection is closed */
                                dnpConnectionEvent(DNP_DISCONNECTED);

                                /* clear the message flags */
                                setdbase(MODBUS, MESSAGE_COMPLETE, 0);
                                setdbase(MODBUS, MESSAGE_FAILED, 0);
                        }
```

```
                        /* save the new state */
                        lastConnectionState = currentConnectionState;
                    }

                    /* release the processor so other tasks can run */
                    release_resource(IO_SYSTEM);
                    release_processor();
                }
}
```

dnpMasterClassPoll

Send DNP Class Poll

Syntax

```
BOOLEAN dnpMasterClassPoll(
      UINT16 slaveAddress,
      UINT16 classFlags
);
```

Description

The `dnpMasterClassPoll` function sends a Class Poll message in DNP, to request the specified data classes from a DNP slave.

`slaveAddress` specifies the DNP station address of the slave.

`classFlags` specifies the classes of data to request. It can contain any combination of the following values; if multiple values are used they should be ORed together:

CLASS0_FLAG, /* request Class 0 Data */

CLASS1_FLAG, /* request Class 1 Data */

CLASS2_FLAG, /* request Class 2 Data */

CLASS3_FLAG /* request Class 3 Data */

The DNP slave (`slaveAddress`) must be configured in the DNP Master Poll Table prior to calling this function.

The function returns TRUE if the DNP class poll message was successfully triggered. It returns FALSE if the specified slave address has not been configured in the DNP Routing Table, or the DNP configuration has not been created.

Notes

This function is only available on the SCADAPack 32 and SCADAPack2.

DNP must be enabled before calling this function in order to create the DNP configuration.

dnpMasterClockSync

Send DNP Clock Synch

Syntax

```
BOOLEAN dnpMasterClockSync(
      UINT16 slaveAddress
);
```

Description

The `dnpMasterClockSync` function sends a Clock Synchronization message in DNP, to a DNP slave.

`slaveAddress` specifies the DNP station address of the slave.

The DNP slave (`slaveAddress`) must be configured in the DNP Master Poll Table prior to calling this function.

The function returns TRUE if the DNP clock sync message was successfully triggered. It returns FALSE if the specified slave address has not been configured in the DNP Routing Table, or the DNP configuration has not been created.

Notes

This function is only available on the SCADAPack 32 and SCADAPack2.

DNP must be enabled before calling this function in order to create the DNP configuration.

dnpPortStatus

Get communication status for a port

Syntax

```
#include <ctools.h>
DNP_PROTOCOL_STATUS dnpPortStatus(
      COM_INTERFACE ifType,
      BOOLEAN clear
      );
```

Description

The `dnpPortStatus` function returns the DNP message statistics for the specified communication port.

`IfType` specifies the communication interface. Valid values are `CIF_Com1, CIF_Com2, CIF_Com3, CIF_Com4,` and `CIF_Lan1.` If `ifType` does not point to a valid communications interface the function has no effect.

If `clear` is TRUE, the DNP message counters are reset to zero after they are read.

dnpReadAddressMappingTableEntry

Read DNP Address Mapping Table entry

Syntax

```
#include <ctools.h>
BOOLEAN dnpReadAddressMappingTableEntry (
      UINT16 index,
      dnpAddressMap_type *pAddressMap
);
```

Description

The **dnpReadAddressMappingTableEntry** function reads an entry from the DNP address mapping table.

pRoute is a pointer to a table entry; it is written by this function.

The return value is TRUE if pAddressMap was successfully written or FALSE otherwise.

Notes

DNP must be enabled before calling this function in order to create the DNP configuration.

dnpReadAddressMappingTableSize

Read DNP Address Mapping Table size

Syntax

```
#include <ctools.h>
UINT16 dnpReadAddressMappingTableSize (void);
```

Description

The **dnpReadAddressMappingTableSize** function reads the total number of entries in the DNP address mapping table.

The function returns the total number of entries in the DNP address mapping table.

Notes

DNP must be enabled before calling this function in order to create the DNP configuration.

dnpReadMasterPollTableEntry

Read DNP Master Poll Table entry

Syntax

```
#include <ctools.h>
BOOLEAN dnpReadMasterPollTableEntry (
      UINT16 index,
      dnpMasterPoll_type *pMasterPoll
);
```

Description

This function reads an entry from the DNP master poll table.

pMasterPoll is a pointer to a table entry; it is written by this function.

The return value is TRUE if pMasterPoll was successfully written or FALSE otherwise.

Notes

DNP must be enabled before calling this function in order to create the DNP configuration.

The function returns the total number of entries in the DNP routing table.

dnpReadMasterPollTableEntryEx

Read DNP Master Poll Table Extended Entry

Syntax

```
BOOLEAN dnpReadMasterPollTableEntryEx (
      UINT16 index,
      DnpMasterPollEx_type *pMasterPoll
);
```

Description

This function is only available on the SCADAPack 32 and SCADAPack2.

This function reads an extended entry from the DNP master poll table.

`pMasterPoll` is a pointer to an extended table entry; it is written by this function.

The return value is TRUE if `pMasterPoll` was successfully written or FALSE otherwise.

Notes

This function is only available on the SCADAPack 32 and SCADAPack2.

DNP must be enabled before calling this function in order to create the DNP configuration.

This function supersedes the `dnpReadMasterPollTableEntry` function.

dnpReadMasterPollTableSize

Read DNP Master Poll Table size

Syntax

```
#include <ctools.h>
UINT16 dnpReadPMasterPollTableSize (void);
```

Description

This function reads the total number of entries in the DNP master poll table.

Notes

DNP must be enabled before calling this function in order to create the DNP configuration.

The function returns the total number of entries in the DNP master poll table.

dnpReadRoutingTableEntry_DialStrings

Read DNP Routing Table Entry Dial Strings

Syntax

```
BOOLEAN dnpReadRoutingTableEntry_DialStrings(
      UINT16 index,
      UINT16 maxPrimaryDialStringLength,
      CHAR *primaryDialString,
      UINT16 maxSecondaryDialStringLength,
      CHAR *secondaryDialString
);
```

Description

This function reads a primary and secondary dial string from an entry in the DNP routing table.

`index` specifies the index of an entry in the DNP routing table.

`maxPrimaryDialStringLength` specifies the maximum length of `primaryDialString` excluding the null-terminator character. The function uses this to limit the size of the returned string to prevent overflowing the storage passed to the function.

`primaryDialString` returns the primary dial string of the target station. It must point to an array of size `maxPrimaryDialStringLength.`

`maxSecondaryDialStringLength` specifies the maximum length of `secondaryDialString` excluding the null-terminator character. The function uses this to limit the size of the returned string to prevent overflowing the storage passed to the function.

`secondaryDialString` returns the secondary dial string of the target station. It must point to an array of size `maxSecondaryDialStringLength.`

The function returns TRUE if the configuration was read and FALSE if an error occurred.

Notes

This function must be used in conjunction with the `dnpReadRoutingTableEntry` function to read a complete entry in the DNP routing table.

dnpReadRoutingTableEntry

Read Routing Table entry

Syntax

```
#include <ctools.h>
BOOLEAN dnpReadRoutingTableEntry(
UINT16 index,
routingTable *pRoute
);
```

Description

This function reads an entry from the routing table.

pRoute is a pointer to a table entry; it is written by this function.

The return value is TRUE if pRoute was successfully written or FALSE otherwise.

Notes

DNP must be enabled before calling this function in order to create the DNP configuration.

See Also

dnpWriteRoutingTableEntry

dnpReadRoutingTableEntryEx

Read Routing Table entry

Syntax

```
#include <ctools.h>
BOOLEAN dnpReadRoutingTableEntryEx(
UINT16 index,
dnpRoutingTableEx entry
);
```

Description

This function reads an extended entry from the DNP routing table.

`index` specifies the index of the entry in the table. Valid values are 0 to the size of the table minus 1.

`pEntry` is a pointer to an extended DNP routing table entry structure. The entry is written to this structure.

The function returns TRUE if the entry was added and FALSE if the index is not valid.

Notes

DNP must be enabled before calling this function in order to create the DNP configuration. Use the `dnpCreateRoutingTable` function to create the routing table and specify its size.

See Also

dnpCreateRoutingTable**,** dnpWriteRoutingTableEntryEx

dnpReadRoutingTableSize

Read Routing Table size

Syntax

```
#include <ctools.h>
UINT16 dnpReadRoutingTableSize (void);
```

Description

This function reads the total number of entries in the routing table.

Notes

DNP must be enabled before calling this function in order to create the DNP configuration.

dnpSaveAI16Config

Save DNP 16-Bit Analog Input Configuration

Syntax

```
#include <ctools.h>
BOOLEAN dnpSaveAI16Config(
      UINT16 point,
      dnpAnalogInput * pAnalogInput
      );
```

Description

The **dnpSaveAI16Config** function sets the configuration of a DNP 16-bit analog input point.

The function has two parameters: the point number; and a pointer to an analog input point configuration structure.

The function returns TRUE if the configuration was written. It returns FALSE if the point number is not valid, if the configuration is not valid, or if DNP configuration has not been created.

Notes

DNP must be enabled before calling this function in order to create the DNP configuration.

Example

See example in the dnpGetConfiguration function section.

dnpSaveAI32Config

Save DNP 32-Bit Analog Input Configuration

Syntax

```
#include <ctools.h>
BOOLEAN dnpSaveAI32Config(
      UINT32 point,
      dnpAnalogInput * pAnalogInput
      );
```

Description

The **dnpSaveAI32Config** function sets the configuration of a DNP 32-bit analog input point.

The function has two parameters: the point number; and a pointer to an analog input point configuration structure.

The function returns TRUE if the configuration was written. It returns FALSE if the point number is not valid, if the configuration is not valid, or if DNP configuration has not been created.

Notes

DNP must be enabled before calling this function in order to create the DNP configuration.

See Also

dnpGetAI32Config

Example

See example in the dnpGetConfiguration function section.

dnpSaveAISFConfig

Save Short Floating Point Analog Input Configuration

Syntax

```
#include <ctools.h>
BOOLEAN dnpSaveAISFConfig (
      UINT16 point,
      dnpAnalogInput *pAnalogInput;
);
```

Description

The **dnpSaveAISFConfig** function sets the configuration of a DNP short floating point analog input point.

The function has two parameters: the point number, and a pointer to a configuration structure.

The function returns TRUE if the configuration was successfully written, or FALSE otherwise (if the point number is not valid, or the configuration is not valid, or if the DNP configuration has not been created).

Notes

DNP must be enabled before calling this function in order to create the DNP configuration.

dnpSaveAO16Config

Save DNP 16-Bit Analog Output Configuration

Syntax

```
#include <ctools.h>
BOOLEAN dnpSaveAO16Config(
      UINT16 point,
      dnpAnalogOutput * pAnalogOutput
      );
```

Description

The **dnpSaveAO16Config** function sets the configuration of a DNP 16-bit analog output point.

The function has two parameters: the point number; and a pointer to an analog output point configuration structure.

The function returns TRUE if the configuration was written. It returns FALSE if the point number is not valid, if the configuration is not valid, or if DNP configuration has not been created.

Notes

DNP must be enabled before calling this function in order to create the DNP configuration.

See Also

dnpGetAO16Config

Example

See example in the dnpGetConfiguration function section.

dnpSaveAO32Config

Save DNP 32-Bit Analog Output Configuration

Syntax

```
#include <ctools.h>
BOOLEAN dnpSaveAO32Config(
      UINT32 point,
      dnpAnalogOutput * pAnalogOutput
      );
```

Description

The **dnpSaveAO32Config** function sets the configuration of a DNP 32-bit analog output point.

The function has two parameters: the point number; and a pointer to an analog output point configuration structure.

The function returns TRUE if the configuration was written. It returns FALSE if the point number is not valid, if the configuration is not valid, or if DNP configuration has not been created.

Notes

DNP must be enabled before calling this function in order to create the DNP configuration.

See Also

dnpGetAO32Config

Example

See example in the dnpGetConfiguration function section.

dnpSaveAOSFConfig

Save Short Floating Point Analog Output Configuration

Syntax

```
#include <ctools.h>
BOOLEAN dnpSaveAOSFConfig (
      UINT16 point,
      dnpAnalogOutput *pAnalogOutput;
);
```

Description

The **dnpSaveAOSFConfig** function sets the configuration of a DNP short floating point analog output point.

The function has two parameters: the point number, and a pointer to a configuration structure.

The function returns TRUE if the configuration was successfully written, or FALSE otherwise (if the point number is not valid, or the configuration is not valid, or if the DNP configuration has not been created).

Notes

DNP must be enabled before calling this function in order to create the DNP

dnpSaveBIConfig

Save DNP Binary Input Configuration

Syntax

```
#include <ctools.h>
BOOLEAN dnpSaveBIConfig(
      UINT16 point,
      dnpBinaryInput * pBinaryInput
      );
```

Description

The **dnpSaveBIConfig** function sets the configuration of a DNP binary input point.

The function has two parameters: the point number; and a pointer to a binary input point configuration structure.

The function returns TRUE if the configuration was written. It returns FALSE if the point number is not valid, if the configuration is not valid, or if DNP configuration has not been created.

Notes

DNP must be enabled before calling this function in order to create the DNP configuration.

See Also

dnpGetBIConfig

Example

See example in the dnpGetConfiguration function section.

dnpSaveBIConfigEx

Write DNP Binary Input Extended Point

Syntax

```
BOOLEAN dnpSaveBIConfigEx(
      UINT16 point,
      dnpBinaryInputEx *pBinaryInput
);
```

Description

This function writes the configuration of an extended DNP Binary Input point.

The function has two parameters: the point number, and a pointer to an extended binary input point configuration structure.

The function returns TRUE if the configuration was successfully written. It returns FALSE if the point number is not valid, if the configuration is not valid, or if the DNP configuration has not been created.

This function supersedes `dnpSaveBIConfig`.

dnpSaveBOConfig

Save DNP Binary Output Configuration

Syntax

```
#include <ctools.h>
BOOLEAN dnpSaveBOConfig(
      UINT16 point,
      dnpBinaryOutput * pBinaryOutput
      );
```

Description

The **dnpSaveBOConfig** function sets the configuration of a DNP binary output point.

The function has two parameters: the point number; and a pointer to a binary output point configuration structure.

The function returns TRUE if the configuration was written. It returns FALSE if the point number is not valid, if the configuration is not valid, or if DNP configuration has not been created.

Notes

DNP must be enabled before calling this function in order to create the DNP configuration.

See Also

dnpGetBOConfig

Example

See example in the dnpGetConfiguration function section.

dnpSaveCI16Config

Save DNP 16-Bit Counter Input Configuration

Syntax

```
#include <ctools.h>
BOOLEAN dnpSaveCI16Config(
      UINT16 point,
      dnpCounterInput * pCounterInput
      );
```

Description

The **dnpSaveCI16Config** function sets the configuration of a DNP 16-bit counter input point.

The function has two parameters: the point number; and a pointer to a counter input point configuration structure.

The function returns TRUE if the configuration was written. It returns FALSE if the point number is not valid, if the configuration is not valid, or if DNP configuration has not been created.

Notes

DNP must be enabled before calling this function in order to create the DNP configuration.

See Also

dnpGetCI16Config

Example

See example in the dnpGetConfiguration function section.

dnpSaveCI32Config

Save DNP 32-Bit Counter Input Configuration

Syntax

```
#include <ctools.h>
BOOLEAN dnpSaveCI32Config(
      UINT32 point,
      dnpCounterInput * pCounterInput
      );
```

Description

The **dnpSaveCI32Config** function sets the configuration of a DNP 32-bit counter input point.

The function has two parameters: the point number; and a pointer to a counter input point configuration structure.

The function returns TRUE if the configuration was written. It returns FALSE if the point number is not valid, if the configuration is not valid, or if DNP configuration has not been created.

Notes

DNP must be enabled before calling this function in order to create the DNP configuration.

See Also

dnpGetCI32Config

Example

See example in the dnpGetConfiguration function section.

dnpSaveConfiguration

Save DNP Configuration

Syntax

```
#include <ctools.h>
BOOLEAN dnpSaveConfiguration(
      dnpConfiguration * pConfiguration
      );
```

Description

The **dnpSaveConfiguration** function sets the DNP configuration.

The function has one parameter, a pointer to a DNP configuration structure.

The function returns TRUE if the configuration was updated and FALSE if an error occurred. No changes are made to any parameters if an error occurs.

Notes

This function must be called before enabling DNP.

The following parameters cannot be changed if DNP is enabled. The function will not make any changes and will return FALSE if this is attempted. The protocol must be disabled in order to make a change involving these parameters.

- `BI_number`
- `BI_cosBufferSize`
- `BI_soeBufferSize`
- `BO_number`
- `CI16_number`
- `CI16_bufferSize`
- `CI32_number`
- `CI32_bufferSize`
- `AI16_number`
- `AI16_reportingMethod`
- `AI16_bufferSize`
- `AI32_number`
- `AI32_reportingMethod`
- `AI32_bufferSize`
- `AO16_number`
- `AO32_number`

The following parameters can be changed when DNP is enabled.

- `masterAddress;`
- `rtuAddress;`
- `datalinkConfirm;`
- `datalinkRetries;`
- `datalinkTimeout;`
- `operateTimeout`
- `applicationConfirm`

- `maximumResponse`
- `applicationRetries`
- `applicationTimeout`
- `timeSynchronization`
- `unsolicited`
- `holdTime`
- `holdCount`

See Also

dnpGetConfiguration

Example

See example in the dnpGetConfiguration function section.

dnpSaveConfigurationEx

Write DNP Extended Configuration

Syntax

```
BOOLEAN dnpSaveConfigurationEx (
      dnpConfigurationEx *pDnpConfigurationEx
);
```

Description

This function writes the extended DNP configuration parameters.

The function has one parameter: a pointer to the DNP extended configuration structure.

The function returns TRUE if the configuration was successfully written, or FALSE otherwise (if the pointer is NULL, or if the DNP configuration has not been created).

Notes

DNP must be enabled before calling this function in order to create the DNP configuration.

This function supersedes the `dnpSaveConfiguration` function.

dnpSendUnsolicitedResponse

Send DNP Unsolicited Response

Syntax

```
BOOLEAN dnpSendUnsolicitedResponse(
      UINT16 classFlags
);
```

Description

The `dnpSendUnsolicitedResponse` function sends an Unsolicited Response message in DNP, with data from the specified classes.

`classFlags` specifies the class or classes of event data to include in the message. It can contain any combination of the following values; if multiple values are used they should be ORed together:

CLASS0_FLAG	enables Class 0 Unsolicited Responses
CLASS1_FLAG	enables Class 1 Unsolicited Responses
CLASS2_FLAG	enables Class 2 Unsolicited Responses
CLASS3_FLAG	enables Class 3 Unsolicited Responses

The function returns TRUE if the DNP unsolicited response message was successfully triggered. It returns FALSE if any of the configured master addresses has not been configured in the DNP Routing Table, or the DNP configuration has not been created.

Notes

DNP must be enabled before calling this function in order to create the DNP configuration.

If no events are pending an empty unsolicited message will be sent.

Example

See the example program DNP Configuration**.**

dnpSearchRoutingTable

Search Routing Table

Syntax

```
#include <ctools.h>
BOOLEAN dnpSearchRoutingTable (
      UINT16 Address
      routingTable *pRoute
);
```

Description

This function searches the routing table for a specific DNP address.

pRoute is a pointer to a table entry; it is written by this function.

The return value is TRUE if pRoute was successfully written or FALSE otherwise.

Notes

DNP must be enabled before calling this function in order to create the DNP configuration.

dnpStationStatus

Get communication status for a remote DNP station

Syntax

```
#include <ctools.h>
DNP_PROTOCOL_STATUS dnpStationStatus(
      UINT16 dnpAddress,
      BOOLEAN clear
      );
```

Description

The `dnpStationStatus` function returns the DNP message statistics for a remote DNP station.

`dnpAddress` is the address of the remote DNP station. Valid values are any DNP station number in the range 1 to 65532.

If `clear` is TRUE, the DNP message counters are reset to zero after they are read.

dnpWriteAddressMappingTableEntry

Write DNP Address Mapping Table Entry

Syntax

```
#include <ctools.h>
BOOLEAN dnpWriteAddressMappingTableEntry (
      UINT16 index,
      UINT16 dnpRemoteStationAddress;
      CHAR dnpObjectType;
      UINT16 dnpRemoteObjectStart;
      UINT16 numberOfPoints;
      UINT16 dnpLocalModbusAddress;
);
```

Description

The **dnpWriteAddressMappingTableEntry** function writes an entry in the DNP address mapping table.

The function returns TRUE if successful, FALSE otherwise.

Notes

DNP must be enabled before calling this function in order to create the DNP configuration.

dnpWriteMasterApplicationLayerConfig

Write DNP Master Application Layer Configuration

Syntax

```
#include <ctools.h>
BOOLEAN dnpWriteMasterApplicationLayerConfig(
UINT16 basePollInterval,
UINT16 mimicMode
);
```

Description

This function writes DNP Master application layer configuration.

Notes

DNP must be enabled before calling this function in order to create the DNP configuration.

The function returns TRUE if successful, FALSE otherwise.

dnpWriteMasterPollTableEntry

Write DNP Master Poll Table Entry

Syntax

```
#include <ctools.h>
BOOLEAN dnpWriteMasterPollTableEntry (
      UINT16 index,
      UINT16 dnpAddress,
      UINT16 class0PollRate;
      UINT16 class1PollRate;
      UINT16 class2PollRate;
      UINT16 class3PollRate;
      UINT16 timeSyncRate;
      UINT16 unsolicitedResponseFlags;
      );
```

Description

This function writes an entry in the DNP master poll table.

Notes

DNP must be enabled before calling this function in order to create the DNP configuration.

The function returns TRUE if successful, FALSE otherwise.

dnpWriteMasterPollTableEntryEx

Write DNP Master Poll Table Extended Entry

Syntax

```
BOOLEAN dnpWriteMasterPollTableEntryEx (
      UINT16 index,
      DnpMasterPollEx_type *pMasterPoll
);
```

Description

This function writes an extended entry in the DNP master poll table.

The function returns TRUE if successful, FALSE otherwise.

Notes

This function is only available on the SCADAPack 32 and SCADAPack2.

DNP must be enabled before calling this function in order to create the DNP configuration.

This function supersedes the `dnpWriteMasterPollTableEntry` function.

dnpWriteRoutingTableEntry_DialString

Write DNP Routing Table Entry Dial Strings

Syntax

```
BOOLEAN dnpWriteRoutingTableEntry_DialStrings(
      UINT16 index,
      UINT16 primaryDialStringLength,
      CHAR *primaryDialString,
      UINT16 secondaryDialStringLength,
      CHAR *secondaryDialString
);
```

Description

This function writes a primary and secondary dial string into an entry in the DNP routing table.

`index` specifies the index of an entry in the DNP routing table.

`primaryDialStringLength` specifies the length of `primaryDialString` excluding the null-terminator character.

`primaryDialString` specifies the dial string used when dialing the target station. This string is used on the first attempt.

`secondaryDialStringLength` specifies the length of `secondaryDialString` excluding the null-terminator character.

`secondaryDialString` specifies the dial string to be used when dialing the target station. It is used for the next attempt if the first attempt fails.

The function returns TRUE if the configuration was written and FALSE if an error occurred.

Notes

This function must be used in conjunction with the `dnpWriteRoutingTableEntry` function to write a complete entry in the DNP routing table.

dnpWriteRoutingTableEntry

Write Routing Table Entry

Syntax

```
#include <ctools.h>
BOOLEAN dnpWriteRoutingTableEntry(
      UINT16 index,
      UINT16 address,
      UINT16 comPort,
      UINT16 retries,
      UINT16 timeout
      );
```

Description

This function writes an entry in the DNP routing table. This function is used to write entries without IP addresses. To create an entry with an IP address, use the `dnpWriteRoutingTableEntryEx` function.

Notes

DNP must be enabled before calling this function in order to create the DNP configuration. Use the `dnpCreateRoutingTable` function to create the routing table and specify its size.

The function returns TRUE if successful, FALSE otherwise.

Example

See the example in the ***dnpGetConfiguration*** section.

dnpWriteRoutingTableEntryEx

Write Routing Table Entry with Extended Information

Syntax

```
#include <ctools.h>
BOOLEAN dnpWriteRoutingTableEntryEx(
      UINT16 index,
      UINT16 address,
      UINT16 comPort,
      UINT16 retries,
      UINT16 timeout,
      IP_ADDRESS ipaddress
      );
```

Description

dnpWriteRoutingTableEntryEx writes an entry in the DNP routing table. This function is used to write entries with IP addresses. To create an entry without an IP address, use the `dnpWriteRoutingTableEntry` function.

Notes

DNP must be enabled before calling this function in order to create the DNP configuration. Use the `dnpCreateRoutingTable` function to create the routing table and specify its size.

The function returns TRUE if successful, FALSE otherwise.

Example

See the example in the ***dnpGetConfiguration*** section.

end_application

Terminates all Application Tasks

Syntax

```
#include <ctools.h>
void end_application(void);
```

Description

The **end_application** function terminates all **APPLICATION** type tasks created with the **create_task** function. Stack space and resources used by the tasks are freed.

Notes

This function is used normally by communication protocols to stop an executing application program, prior to loading a new program into memory.

See Also

end_group, end_task

end_group

Terminates all Tasks in a Task Group

Syntax

```
#include <ctools.h>
void end_group(UINT16 taskGroup);
```

Description

The **end_group** function terminates all tasks of the specified type. This function should only be used with taskGroups of APPLICATION_GROUP_0 – APPLICATION_GROUP_9. Stack space and resources used by the tasks are freed.

Notes

This function is used normally by communication protocols to stop an executing application program.

See Also

end_application, end_task

end_task

Terminate a Task

Syntax

```
#include <ctools.h>
void end_task(UINT16 task_ID);
```

Description

The **end_task** function terminates the task specified by *task_ID*. Stack space and resources used by the task are freed. The **end_task** function terminates any type task.

See Also

end_application, end_group

endTimedEvent

Terminate Signaling of a Regular Event

Syntax

```
#include <ctools.h>
UINT16 endTimedEvent(UINT16 event);
```

Description

This **endTimedEvent** function cancels signaling of a timed event, initialized by the startTimedEvent function.

The function returns TRUE if the event signaling was canceled.

The function returns FALSE if the event number is not valid, or if the event was not previously initiated with the startTimedEvent function. The function has no effect in these cases.

Notes

Valid events are numbered 0 to RTOS_EVENTS - 1. Any events defined in ctools.h are not valid events for use in an application program.

Example

See the examples for **startTimedEvent**.

See Also

startTimedEvent

enronInstallCommandHandler

Installs handler for Enron Modbus commands

Syntax

```
#include <ctools.h>
void enronInstallCommandHandler(
      UINT16 (* function)(
            UINT16   length,
            UCHAR  * pCommand,
            UINT16   responseSize,
            UINT16 * pResponseLength,
            UCHAR  * pResponse
            )
      );
```

Description

This function installs a handler function for Enron Modbus commands. The protocol driver calls this handler function each time a command is received for the Enron Modbus station.

`function` is a pointer to the handler function. If function is NULL the handler is disabled.

The function has no return value.

Notes

The application must disable the handler when the application ends. This prevents the protocol driver from calling the handler while the application is stopped. Call the `enronInstallCommmandHandler` with a NULL pointer. The usual method is to create a task exit handler function to do this. See the example below for details.

The handler function has five parameters.

- `length` is the number of characters in the command message.
- `pCommand` is a pointer to the command message. The first byte in the message is the function code, followed by the Enron Modbus message. See the Enron Modbus protocol specification for details on the message formats.
- `responseSize` is the size of the response buffer in characters.
- `pResponseLength` is a pointer to a variable that will hold the number of characters in the response. If the handler returns TRUE, it must set this variable.
- `pResponse` is a pointer to a buffer that will hold the response message. The buffer size is `responseSize` characters. The handler must not write beyond the end of the buffer. If the handler returns TRUE, it must set this variable. The data must start with the function code and end with the last data byte. The protocol driver will add the station address, checksum, and message framing to the response.

The handler function returns the following values.

Value	Description
NORMAL	Indicates protocol handler should send a normal response message. Data are returned using pResponse and pResponseLength.
ILLEGAL_FUNCTION	Indicates protocol handler should send an Illegal Function exception response message. This response should be used when the function code in the command is not recognized.
ILLEGAL_DATA_ADDRESS	Indicates protocol handler should send an Illegal Data Address exception response message. This response should be used when the data address in the command is not recognized.
ILLEGAL_DATA_VALUE	Indicates protocol handler should send an Illegal Data Value exception response message. This response should be used when invalid data is found in the command.

If the function returns NORMAL then the protocol driver sends the response message in the buffer pointed to by pResponse. If the function returns an exception response protocol driver returns the exception response to the caller. The buffer pointed to by pResponse is not used.

Example

This program installs a simple handler function.

```
#include <ctools.h>

/* ------------------------------------------------------
   This function processes Enron Modbus commands.
   ------------------------------------------------------ */
UINT16 commandHandler(
      UINT16   length,
      UCHAR  * pCommand,
      UINT16   responseSize,
      UINT16 * pResponseLength,
      UCHAR  * pResponse
      )
{
      UCHAR command;
      UINT16 result;

      /* if a command byte was received */
      if (length >= 1)
      {
            /* get the command byte */
            command = pCommand[0];
            switch (command)
            {
            /* read unit status command */
            case 7:
                  /* if the response buffer is large enough */
                  if (responseSize > 2)
                  {
                        /* build the response header */
```

```
                        pResponse[0] = pCommand[0];

                        /* set the unit status */
                        pResponse[1] = 17;

                        /* set response length */
                        *pResponseLength = 2;

                        /* indicate the command worked */
                        result = NORMAL;
                  }
                  else
                  {
                        /* buffer is to small to respond */
                        result = ILLEGAL_FUNCTION;
                  }
                  break;

            /* add cases for other commands here */

            default:
                  /* command is invalid */
                  result = ILLEGAL_FUNCTION;
            }
      }
      else
      {
            /* command is too short so return error */
            result = ILLEGAL_FUNCTION;
      }
      return result;
}

/* ------------------------------------------------------
   This function unhooks the protocol handler when the
   main task ends.
   ------------------------------------------------------ */
void mainExitHandler(void)
{
      /* unhook the handler function */
      enronInstallCommandHandler(NULL);
}

int main(void)
{
      TASKINFO thisTask;

      /* install handler to execute when this task ends */
      thisTask = getTaskInfo(0);
      installExitHandler(thisTask.taskID, (FUNCPTR)
mainExitHandler);

      /* install handler for Enron Modbus */
      enronInstallCommandHandler(commandHandler);

      /* infinite loop of main task */
      while (TRUE)
```

```
        {
                /* add application code here */
        }
}
```

ethernetGetIP

Get Ethernet Controller TCP/IP Settings

Syntax

```
#include <ctools.h>
void ethernetGetIP( IP_SETTINGS * pIPSettings );
```

Description

The **ethernetGetIP** function copies the Ethernet controller TCP/IP settings into the structure pointed to by *pIPSettings*. The structure IP_SETTINGS is described in the *Structures and Types* section.

See Also

ethernetSetIP

ethernetGetMACAddress

Get Ethernet Controller MAC address

Syntax

```
#include <ctools.h>
void ethernetGetMACAddress( CHAR * pMAC );
```

Description

The **ethernetGetMACAddress** function copies the Ethernet controller MAC address to the array pointed to by *pMAC. pMAC* must point to an array of 6 bytes.

ethernetSetIP

Set Ethernet Controller TCP/IP Setting

Syntax

```
#include <ctools.h>
BOOLEAN ethernetSetIP( IP_SETTINGS * pIPSettings );
```

Description

The **ethernetSetIP** function copies the settings pointed to by pIPSettings to the Ethernet controller settings. If the settings are different from the current settings, the Ethernet interface is closed and re-opened with the new settings. When the Ethernet interface is closed all active connections through this interface are closed.

The structure IP_SETTINGS is described in the *Structures and Types* section. If there is an invalid setting, FALSE is returned and the settings are not saved; otherwise TRUE is returned.

To save these settings with the controller settings in flash memory so that they are loaded on controller reset, call **flashSettingsSave** as shown below.

```
request_resource(FLASH_MEMORY);
flashSettingsSave(CS_PERMANENT);
release_resource(FLASH_MEMORY);
```

flashSettingsLoad

Load Controller Settings from Flash

Syntax

```
#include <ctools.h>
BOOLEAN flashSettingsLoad(UINT32 areaFlags);
```

Description

This function loads the controller settings in the indicated area or areas from flash memory. Settings in other areas are not affected.

The function has one parameter, **areaFlags**, indicating which areas to read from flash. A sum of more than one area may be selected.

If an unsupported flag is set, the flag has no effect. If there is no supported flag set (e.g. areaFlags=0), nothing is done.

The function has no return value.

See the function **flashSettingsSave** for a list of valid flags.

Notes

The FLASH_MEMORY resource must be requested before calling this function.

flashSettingsSave

Save Controller Settings to Flash

Syntax

```
#include <ctools.h>
BOOLEAN flashSettingsSave(UINT32 areaFlags);
```

Description

This function stores the controller settings in the indicated area or areas to flash memory. Settings in other areas are not affected.

The function has one parameter, **areaFlags**, indicating which areas to store into flash. A sum of more than one area may be selected.

The function returns TRUE if all the settings were stored and FALSE if there was an error writing to flash.

If an unsupported flag is set, the flag has no effect. If there is no supported flag set (e.g. areaFlags=0), all current settings are saved again.

Valid flags are listed below and defined in ctools.h.

Area Flag	Loaded on Reset	Controller Settings in this Area
CS_ETHERNET	always	Ethernet MAC address
CS_OPTIONS	always	Controller factory options.
CS_PERMANENT	Saved settings loaded on Service and Run Boot. Replaced with default settings on Cold Boot.	Controller type, IP address, Gateway, Network mask, IP Configuration mode, Lock state and password, I/O System settings
CS_RUN	Saved settings loaded on Run Boot. Default settings loaded on Service Boot. Replaced with default settings on Cold Boot.	Serial port settings, Serial protocol settings, Modbus/TCP settings, HART I/O settings, LED power settings,

Notes

The FLASH_MEMORY resource must be requested before calling this function.

forceLed

Set State of Force LED

Syntax

```
#include <ctools.h>
void forceLed(UINT16 state);
```

Description

The **forceLed** function sets the state of the FORCE LED. *state* may be either LED_ON or LED_OFF.

Notes

The FORCE LED is used to indicate forced I/O. Use this function with caution in application programs.

freeMemory

Free Non-Volatile Dynamic Memory

Syntax

```
#include <ctools.h>
void freeMemory(void *pMemory);
```

Description

The **freeMemory** function returns the specified memory to the system memory pool. The specified memory to be returned or freed must have been allocated by a previous call to the function **allocateMemory.**

The function has one argument: a pointer to the memory to be freed.

Notes

The DYNAMIC_MEMORY resource must be requested before calling this function.

The allocation of memory and the allocated memory are non-volatile.

Pointers to non-volatile dynamic memory must be statically allocated in a non-volatile data section. Otherwise they will be initialised at reset and the non-volatile dynamic memory will be lost. See the example for the function **allocateMemory** which demonstrates how to create a non-volatile data section to save pointers to non-volatile dynamic memory.

See Also

allocateMemory

getABConfiguration

Get DF1 Protocol Configuration

Syntax

```
#include <ctools.h>
struct ABConfiguration *getABConfiguration(FILE *stream, struct
ABConfiguration *ABConfig);
```

Description

The **getABConfiguration** function gets the DF1 protocol configuration parameters for the *stream*. If *stream* does not point to a valid serial port the function has no effect. *ABConfig* must point to a DF1 protocol configuration structure.

The **getABConfiguration** function copies the DF1 configuration parameters into the *ABConfig* structure and returns a pointer to it.

See Also

setABConfiguration

Example

This program displays the DF1 configuration parameters for **com1**.

```
#include <ctools.h>

int main(void)
{
   struct ABConfiguration ABConfig;

   getABConfiguration(com1, &ABConfig);
   fprintf(com1,"Min protected address:     %u\r\n",
      ABConfig.min_protected_address);
   fprintf(com1,"Max protected address:     %u\r\n",
      ABConfig.max_protected_address);
}
```

getclock

Read the Real Time Clock

Syntax

```
#include <ctools.h>
void getclock(TIME * time);
```

Description

The **getclock** function reads the time and date from the real time clock hardware.

The **getclock** function copies the time and date information to the TIME structure pointed to by *time*.

Notes

The time format returned by the **getclock** function is not compatible with the standard UNIX style functions.

The IO_SYSTEM resource must be requested before calling this function.

See Also

getClockTime, setclock

Example

This program displays the current date and time.

```
#include <ctools.h>
main(void)
{
       TIME now;

       request_resource(IO_SYSTEM);
       getclock(&now);        /* read the clock */
       release_resource(IO_SYSTEM);
       fprintf(com1,"%2d/%2d/%2d", now.day, now.month, now.year);
       fprintf(com1,"%2d:%2d\r\n",now.hour, now.minute);
}
```

getClockAlarm

Read the Real Time Clock Alarm Settings

Syntax

```
#include <ctools.h>
ALARM_SETTING getClockAlarm(void);
```

Description

The **getClockAlarm** function returns the alarm setting in the real time clock.

Notes

The IO_SYSTEM resource must be requested before calling this function.

See Also

setClockAlarm

getClockTime

Read the Real Time Clock

Syntax

```
#include <ctools.h>
void getClockTime(INT32 * pDays, INT32 * pHundredths);
```

Description

The getClockTime function reads the real time clock and returns the value as the number of whole days since 01/01/1997 and the number of hundredths of a second since the start of the current day. The function works for years from 01/01/1997 to 12/31/2099 then rolls over.

The function has two parameters: a pointer to the variable to hold the days and a pointer to a variable to hold the hundredths of a second.

The function has no return value.

Notes

The IO_SYSTEM resource must be requested before calling this function.

See Also

getclock

getControllerID

Get Controller ID

Syntax

```
#include <ctools.h>
void getControllerID(CHAR * pID)
```

Description

This function writes the Controller ID to the string pointed to by *pID*. The Controller ID is a unique ID for the controller set at the factory. The pointer *pID* must point to a character string of length CONTROLLER_ID_LEN.

Example

This program displays the Controller ID.

```
#include <ctools.h>

int main(void)
{
       char     ctlrID[CONTROLLER_ID_LEN];
       UINT16 index;

       getControllerID(ctlrID);

       fprintf(com1, "\r\nController ID : ");
       for (index=0; index<CONTROLLER_ID_LEN; index++)
       {
          fputc(ctlrID[index], com1);
       }
}
```

getForceFlag

Get Force Flag State for a Register (TelePACE firmware only)

Syntax

```
#include <ctools.h>
BOOLEAN getForceFlag(UINT16 type, UINT16 address, UINT16 *value);
```

Description

The **getForceFlag** function copies the value of the force flag for the specified database register into the integer pointed to by *value*. The valid range for *address* is determined by the database addressing *type*.

The force flag value is either 1 or 0, or a 16-bit mask for LINEAR digital addresses.

If the *address* or addressing *type* is not valid, FALSE is returned and the integer pointed to by *value* is 0; otherwise TRUE is returned. The table below shows the valid address types and ranges.

Type	Address Ranges	Register Size
MODBUS	00001 to NUMCOIL 10001 to 10000 + NUMSTATUS 30001 to 30000 + NUMINPUT 40001 to 40000 + NUMHOLDING	1 bit 1 bit 16 bit 16 bit
LINEAR	0 to NUMLINEAR-1	16 bit

Notes

Force Flags are not modified when the controller is reset. Force Flags are in a permanent storage area, which is maintained during power outages.

Refer to the *I/O Database and Register Assignment* chapter for more information.

See Also

setForceFlag**,** clearAllForcing**,** overrideDbase

Example

This program obtains the force flag state for register 40001, for the 16 status registers at linear address 302 (i.e. registers 10737 to 10752), and for the holding register at linear address 1540 (i.e. register 40005).

```
#include <ctools.h>

int main(void)
{
       UINT 16 flag, bitmask;

       getForceFlag(MODBUS, 40001, &flag);
       getForceFlag(LINEAR, 302, &bitmask);
       getForceFlag(LINEAR, 1540, &flag);
}
```

getForceLed

Get status of Force LED

Syntax

```
#include <ctools.h>
UINT16 getForceLed( void )
```

Description

The **getForceLed** function returns the status of the Force LED. It returns TRUE if the LED is ON and FALSE if the LED is OFF.

See Also

forceLed

getHardwareInformation

Obtains the hardware type and version

Syntax

```
#include <ctools.h>
BOOLEAN getHardwareInformation(UCHAR* majorVersion, UCHAR* minorVersion,
UCHAR* hardwareType);
```

Description

The **getHardwareInformation** function will place the major version of the hardware into the memory pointed to by `majorVersion`, the minor version of the hardware into the `minorVersion`, and the hardware type in the memory pointed to by `hardwareType`. Refer to the macros starting with HT_ for the various hardware types.

The function returns TRUE if the hardware version and type was placed in the passed variables. Otherwise FALSE is returned.

Notes

This function is currently only supported on the SCADAPack2

getIOErrorIndication

Get I/O Module Error Indication

Syntax

```
#include <ctools.h>
BOOLEAN getIOErrorIndication(void);
```

Description

The **getIOErrorIndication** function returns the state of the I/O module error indication. TRUE is returned if the I/O module communication status is currently reported in the controller status register and Status LED. FALSE is returned if the I/O module communication status is not reported.

Notes

Refer to the ***5203/4 System Manual, SCADAPack 32 System Manual,*** or the ***SCADAPack2 System Manual*** for further information on the Status LED and Status Output.

See Also

setIOErrorIndication

getOutputsInStopMode

Get Outputs In Stop Mode (TelePACE firmware only)

Syntax

```
#include <ctools.h>
void getOutputsInStopMode(
   BOOLEAN *holdDoutsOnStop,
   BOOLEAN *holdAoutsOnStop
   );
```

Description

The **getOutputsInStopMode** function copies the values of the output control flags into the integers pointed to by *doutsInStopMode* and *aoutsInStopMode*.

If the value pointed to by *holdDoutsOnStop* is TRUE, then digital outputs are held at their last state when the Ladder Logic program is stopped.

If the value pointed to by *holdDoutsOnStop* is FALSE, then digital outputs are turned OFF when the Ladder Logic program is stopped.

If the value pointed to by *holdAoutsOnStop* is TRUE, then analog outputs are held at their last value when the Ladder Logic program is stopped.

If the value pointed to by *holdAoutsOnStop* is FALSE, then analog outputs go to zero when the Ladder Logic program is stopped.

See Also

setOutputsInStopMode

Example

See the example for **setOutputsInStopMode** function.

getpeername

Syntax

```
#include <ctools.h>
int getpeername
(
      int socketDescriptor,
      Struct sockaddr * fromAddressPtr,
      int * addressLengthPtr
);
```

Function Description

This function returns to the caller the IP address / Port number of the remote system that the socket is connected to.

Parameter Description

socketDescriptor	The socket descriptor that we wish to obtain this information about.
fromAddressPtr	A pointer to the address structure that we wish to store this information into.
addressLengthPtr	The length of the address structure.

Returns

0	Success
-1	An error occurred

getpeername can fail for any of the following reasons:

EBADF	*socketDescriptor* is not a valid descriptor.
ENOTCONN	The socket is not connected.
EINVAL	One of the passed parameters is not valid.

getPortCharacteristics

Get Serial Port Characteristics

Syntax

```
#include <ctools.h>
BOOLEAN getPortCharacteristics(FILE *stream, PORT_CHARACTERISTICS
   *pCharacteristics);
```

Description

The **getPortCharacteristics** function gets information about features supported by the serial port pointed to by *stream*. If *stream* does not point to a valid serial port the function has no effect and FALSE is returned; otherwise TRUE is returned.

The **getPortCharacteristics** function copies the serial port characteristics into the structure pointed to by *pCharacteristics*.

Notes

Refer to the **Overview of Functions** section for detailed information on serial ports.

Refer to the **Structures and Types** section for a description of the fields in the PORT_CHARACTERISTICS structure.

See Also

get_port

Example

```
#include <ctools.h>
int main(void)
{
      PORT_CHARACTERISTICS options;

      getPortCharacteristics(com3, &options);
      fprintf(com1, "Dataflow options: %d\r\n", options.dataflow);
      fprintf(com1, "Protocol options: %d\r\n", options.protocol);
}
```

get_port

Get Serial Port Configuration

Syntax

```
#include <ctools.h>
struct pconfig *get_port(UCHAR port, struct pconfig *settings);
```

Description

The **get_port** function gets the serial port configuration for the *port*. If *port* is not a valid serial port the function has no effect.

The **get_port** function copies the serial port settings into the structure pointed to by *settings* and returns a pointer to the structure.

Notes

Refer to the **Overview of Functions** section for detailed information on serial ports.

Refer to the **Structure and Types** section for a description of the fields in the *pconfig* structure.

See Also

set_port

Example

```
#include <ctools.h>

int main(void)
{
      struct pconfig settings;

      get_port(com1, &settings);
      fprintf(com1,"Baud rate: %d\r\n", settings.baud);
      fprintf(com1,"Duplex:    %d\r\n", settings.duplex);
}
```

getPowerMode

Get Current Power Mode

Syntax

```
#include <ctools.h>
BOOLEAN getPowerMode(UCHAR* cpuPower, UCHAR* lan, UCHAR* usbPeripheral,
UCHAR* usbHost);
```

Description

The **getPowerMode** function places the current state of the CPU, LAN, USB peripheral port, and USB host port in the passed parameters. The following table lists the possible return values and their meaning.

Macro	*Meaning*
PM_CPU_FULL	The CPU is set to run at full speed
PM_CPU_REDUCED	The CPU is set to run at a reduced speed
PM_CPU_SLEEP	The CPU is set to sleep mode
PM_LAN_ENABLED	The LAN is enabled
PM_LAN_DISABLED	The LAN is disabled
PM_USB_PERIPHERAL_ENABLED	The USB peripheral port is enabled
PM_USB_PERIPHERAL_DISABLED	The USB peripheral port is disabled
PM_USB_HOST_ENABLED	The USB host port is enabled
PM_USB_HOST_DISABLED	The USB host port is disabled
PM_UNAVAILABLE	The status of the device could not be read.

TRUE is returned if the values placed in the passed parameters are valid, otherwise FALSE is returned.

The application program may set the current power mode with the s**etPowerMode** function.

See Also

setPowerMode, setWakeSource, getWakeSource

getProgramStatus

Get Program Status Flag

Syntax

```
#include <ctools.h>
UINT16 getProgramStatus(FUNCPTR entryPoint);
```

Description

The **getProgramStatus** function returns the application program status flag of the program specified by entryPoint. The passed parameter should always be the function `main`. The status flag is set to **NEW_PROGRAM** when the C program downloaded to the controller from the program loader. The status flag is set to **PROGRAM_NOT_LOADED** when the C program is erased.

The application program may modify the status flag with the **setProgramStatus** function.

See Also

setProgramStatus

Example

See the Get Program Status Example in the Examples section.

get_protocol

Get Protocol Configuration

Syntax

```
#include <ctools.h>
struct prot_settings *get_protocol(UCHAR port, struct prot_settings
   *settings);
```

Description

The **get_protocol** function gets the communication protocol configuration for the *port*. If *port* does not point to a valid serial port the function has no effect. *settings* must point to a protocol configuration structure, *prot_settings*.

The **get_protocol** function copies the protocol settings into the structure pointed to by *settings* and returns a pointer to that structure.

Refer to the *ctools.h* file for a description of the fields in the *prot_settings* structure.

Refer to the **Overview of Functions** section for detailed information on communication protocols.

See Also

set_protocol

Example

This program displays the protocol configuration for **com1**.

```
#include <ctools.h>

int main(void)
{
       struct prot_settings settings;

       get_protocol(com1, &settings);
       fprintf(com1,"Type:     %d\r\n", settings.type);
       fprintf(com1,"Station:  %d\r\n", settings.station);
       fprintf(com1,"Priority: %d\r\n", settings.priority);
}
```

getProtocolSettings

Get Protocol Extended Addressing Configuration

Syntax

```
#include <ctools.h>
BOOLEAN getProtocolSettings(
      UCHAR port,
      PROTOCOL_SETTINGS * settings
      );
```

Description

The getProtocolSettings function reads the protocol parameters for a serial port. This function supports extended addressing.

The function has two parameters: *port* is one of com1, com2 or com3; and *settings,* a pointer to a PROTOCOL_SETTINGS structure. Refer to the description of the structure for an explanation of the parameters.

The function returns **TRUE** if the structure was changed. It returns **FALSE** if the stream is not valid.

Notes

Extended addressing is available on the Modbus RTU and Modbus ASCII protocols only. See the *TeleBUS Protocols User Manual* for details.

Refer to the ***TeleBUS Protocols User Manual*** section for detailed information on communication protocols.

See Also

setProtocolSettings**,** get_protocol

Example

This program displays the protocol configuration for com1.

```
#include <ctools.h>

int main(void)
{
       PROTOCOL_SETTINGS settings;

       if (getProtocolSettings(com1, &settings)
       {
           fprintf(com1,"Type: %d\r\n", settings.type);
           fprintf(com1,"Station: %d\r\n", settings.station);
           fprintf(com1,"Address Mode: %d\r\n", settings.mode);
           fprintf(com1,"SF Messaging: %d\r\n", settings.SFMessaging);
           fprintf(com1,"Priority: %d\r\n", settings.priority);
       }
       else
       {
           fprintf(com1,“Serial port is not valid\r\n”);
       }
}
```

getProtocolSettingsEx

Reads extended protocol settings for a serial port

Syntax

```
#include <ctools.h>
BOOLEAN getProtocolSettingsEx(
      UCHAR port,
      PROTOCOL_SETTINGS_EX * pSettings
      );
```

Description

The `setProtocolSettingsEx` function sets protocol parameters for a serial port. This function supports extended addressing and Enron Modbus parameters.

The function has two arguments:

- `port` specifies the serial port. It is one of com1, com2 or com3.
- `pSettings` is a pointer to a `PROTOCOL_SETTINGS_EX` structure. Refer to the description of the structure for an explanation of the parameters.

The function returns `TRUE` if the settings were retrieved. It returns `FALSE` if the stream is not valid.

Notes

Extended addressing and the Enron Modbus station are available on the Modbus RTU and Modbus ASCII protocols only. See the *TeleBUS Protocols User Manual* for details.

See Also

setProtocolSettingsEx, setProtocolSettings, start_protocol, get_protocol, get_protocol_status, set_protocol, modemNotification

Example

This program displays the protocol configuration for com1.

```
#include <ctools.h>
int main(void)
{
      PROTOCOL_SETTINGS_EX settings;
      if (getProtocolSettingsEx(com1, &settings)
      {
            fprintf(com1,"Type: %d\r\n", settings.type);
            fprintf(com1,"Station: %d\r\n", settings.station);
            fprintf(com1,"Address Mode: %d\r\n", settings.mode);
            fprintf(com1,"SF: %d\r\n", settings.SFMessaging);
            fprintf(com1,"Priority: %d\r\n", settings.priority);
            fprintf(com1,"Enron: %d\r\n", settings.enronEnabled);
            fprintf(com1,"Enron station: %d\r\n",
                  settings.enronStation);
      }
```

```
        else
        {
              fprintf(com1,“Serial port is not valid\r\n”);
        }
}
```

get_protocol_status

Get Protocol Information

Syntax

```
#include <ctools.h>
struct prot_status get_protocol_status(FILE *stream);
```

Description

The **get_protocol_status** function returns the protocol error and message counters for *stream*. If *stream* does not point to a valid serial port the function has no effect.

Refer to the **Overview of Functions** section for detailed information on communication protocols.

See Also

clear_protocol_status

Example

This program displays the checksum error counter for **com2**.

```
#include <ctools.h>

int main(void)
{
       struct prot_status status;

       status = get_protocol_status(com2);
       fprintf(com1,"Checksum: %d\r\n", status.checksum_errors);
}
```

getSFTranslation

Read Store and Forward Translation

Syntax

```
#include <ctools.h>
void getSFTranslation(UINT16 index, SF_TRANSLATION * pTranslation);
```

Description

Instead of using the **getSFTranslation** function use the **getSFTranslationEx** function, which supports translations with a timeout and authentication.

The **getSFTranslation** function copies the entry from the store and forward translation table at *index* to the structure pointed to by *pTranslation*. If *index* is invalid, a disabled table entry is copied. The disabled table entry has both station fields set to 65535.

The SF_TRANSLATION structure is described in the *Structures and Types* section. manual.

Notes

The ***TeleBUS Protocols User Manual*** describes store and forward messaging mode.

See Also

setSFTranslation, clearSFTranslationTable, checkSFTranslationTable

getSFTranslationEx

Read Store and Forward Translation Method 2

Syntax

```
#include <ctools.h>
void getSFTranslationEx(UINT16 index, SF_TRANSLATION_EX * pTranslation,
   UCHAR* userName);
```

Description

The **getSFTranslationEx** function copies the entry from the store and forward translation table at *index* to the structure pointed to by *pTranslation*. If *index* is invalid, a disabled table entry is copied. The disabled table entry has both station fields set to 65535. If the *userName* parameter is non-NULL then the user name used for authentication purposes will be copied into the array pointed to by *userName*. userName must point to an array of 16 unsigned characters.

The SF_TRANSLATION_EX structure supports a timeout and is described in the *Structures and Types* section.

Notes

The ***TeleBUS Protocols User Manual*** describes the store and forward messaging mode.

See Also

setSFTranslationEx**,** clearSFTranslationTable, checkSFTranslationTable

getsockname

Syntax

```
#include <ctools.h>
int getsockname
(
      int socketDescriptor,
      struct sockaddr * myAddressPtr,
      int * addressLengthPtr
);
```

Function Description

This function returns to the caller the Local IP Address / Port Number that we are using on a given socket.

Parameters

socketDescriptor	The socket descriptor that we wish to inquire about.
myAddressPtr	The pointer to the address structure where the address information will be stored.
*addressLengthP*tr	The length of the address structure.

Returns

0	Success
-1	An error occurred

getsockname can fail for any of the following reasons:

EBADF	*socketDescriptor* is not a valid descriptor.
EINVAL	One of the passed parameters is not valid.

getsockopt

Syntax

```
#include <ctools.h>
int getsockopt
(
      int socketDescriptor,
      int protocolLevel,
      int optionName,
      char * optionValuePtr,
      int * optionLengthPtr
);
```

Function Description

getsockopt is used retrieve options associated with a socket. Options may exist at multiple protocol levels; they are always present at the uppermost “socket” level. When manipulating socket options, the level at which the option resides and the name of the option must be specified. To manipulate options at the “socket” level, *protocolLevel* is specified as SOL_SOCKET. To manipulate options at any other level, *protocolLevel* is the protocol number of the protocol that controls the option. For example, to indicate that an option is to be interpreted by the TCP protocol, *protocolLevel* is set to the TCP protocol number. For **getsockopt**, the parameters *optionValuePtr* and *optionLengthPtr* identify a buffer in which the value(s) for the requested option(s) are to be returned. For **getsockopt**, *optionLengthPtr* is a value-result parameter, initially containing the size of the buffer pointed to by *optionValuePt*r, and modified on return to indicate the actual size of the value returned. *optionName* and any specified options are passed un-interpreted to the appropriate protocol module for interpretation. The include file <ctools.h> contains definitions for the options described below. Options vary in format and name. Most socket-level options take an int for *optionValuePt*r. SO_LINGER uses a *struct linger* parameter that specifies the desired state of the option and the linger interval (see below). struct linger is defined in <ctools.h>. *struct linger* contains the following members:

l_onoff on = 1 / off = 0

l_linger linger time, in seconds.

The following options are recognized at the socket level:

SOL_SOCKET *protocolLevel* options

SO_ACCEPTCONN	Enable/disable listening for connections. listen turns on this option.
SO_DONTROUTE	Enable/disable routing bypass for outgoing messages. Default 0.
SO_KEEPALIVE	Enable/disable keep connections alive. Default 0 (disable)
SO_OOBINLINE	Enable/disable reception of out-of-band data in band. Default is 0.
SO_REUSEADDR	Enable/disable local address reuse. Default 0 (disable).
SO_RCVLOWAT	The low water mark for receiving.
SO_SNDLOWAT	The low water mark for sending.
SO_RCVBUF	The buffer size for input. Default is 8192 bytes.
SO_SNDBUF	The buffer size for output. Default is 8192 bytes.

SO_REUSEADDR indicates that the rules used in validating addresses supplied in a **bind** call should allow reuse of local addresses. SO_KEEPALIVE enables the periodic transmission of messages (every 2 hours) on a connected socket. If the connected party fails to respond to these messages, the connection is considered broken. SO_DONTROUTE indicates that outgoing messages should bypass the standard routing facilities. Instead, messages are directed to the appropriate network interface according to the network portion of the destination address. SO_LINGER controls the action taken when unsent messages are queued on a socket and a close on the socket is performed. If the socket promises reliable delivery of data and SO_LINGER is set, the system will block the process on the close of the socket attempt until it is able to transmit the data or until it decides it is unable to deliver the information (a timeout period, termed the linger interval, is specified in the **setsockopt** call when SO_LINGER is requested). If SO_LINGER is disabled and a close on the socket is issued, the system will process the close of the socket in a manner that allows the process to continue as quickly as possible. The option SO_BROADCAST requests permission to send broadcast datagrams on the socket. With protocols that support out-of-band data, the SO_OOBINLINE option requests that out-of-band data be placed in the normal data input queue as received; it will then be accessible with **recv** call without the MSG_OOB flag. SO_SNDBUF and SO_RCVBUF are options that adjust the normal buffer sizes allocated for output and input buffers, respectively. The buffer size may be increased for high-volume connections or may be decreased to limit the possible backlog of incoming data. The Internet protocols place an absolute limit of 64 Kbytes on these values for UDP and TCP sockets (in the default mode of operation).

The following options are recognized at the IP level.

IP_PROTOIP *protocolLevel* options

IP_MULTICAST_IF	Get the configured IP address that uniquely identifies the outgoing interface for multicast datagrams sent on this socket. A zero IP address parameter indicates that we want to reset a previously set outgoing interface for multicast packets sent on that socket.
IP_MULTICAST_TTL	Get the default IP TTL for outgoing multicast datagrams.
IP_TOS	IP type of service. Default 0
IP_TTL	IP Time To Live in seconds. Default 64.

The following options are recognized at the TCP level.

IP_PROTOTCP *protocolLevel* options

TCP_MAXSEG	Get the maximum TCP segment size sent on the network. Note that the TCP_MAXSEG value is the maximum amount of data (including TCP options, but not the TCP header) that can be sent per segment to the peer. i.e. the amount of user data sent per segment is the value given by the TCP_MAXSEG option minus any enabled TCP option (for example 12 bytes for a TCP time stamp option) . Default is IP MTU minus 40 bytes.
TCP_NODELAY	If this option value is non-zero, the Nagle algorithm that buffers the sent data inside the TCP is disabled. Useful to allow client's TCP to send small packets as soon as possible (like mouse clicks). Default 0.

Parameters

socketDescriptor	The socket descriptor to get the option from.

protocolLevel	The protocol to get the option from. See below.
optionName	The option to get. See above and below.
optionValuePtr	The pointer to a user variable into which the option value is returned. User variable is of data type described below.
optionLengthPtr	Pointer to the size of the user variable, which is the size of the option data type, described below. It is a value-result parameter, and the user should set the size prior to the call.
SOL_SOCKET	Socket level protocol
IP_PROTOIP	IP level protocol
IP_PROTOTCP	TCP level protocol.

Protocol Level	**Option Name**	**Option data type**	**Option value**
SOL_SOCKET	SO_ACCEPTCONN	int	0 or 1
	SO_DONTROUTE	int	0 or 1
	SO_KEEPALIVE	int	0 or 1
	SO_LINGER	struct linger	
	SO_OOBINLINE	int	0 or 1
	SO_RCVBUF	unsigned long	
	SO_RCVLOWAT	unsigned long	
	SO_REUSEADDR	int	0 or 1
	SO_SNDBUF	unsigned long	
	SO_SNDLOWAT	unsigned long	
IP_PROTOIP	IP_MULTICAST_IF	struct in_addr	
	IP_MULTICAST_TTL	unsigned char	
	IP_TOS	unsigned char	
	IP_TTL	unsigned char	
IP_PROTOTCP	TCP_MAXSEG	int	
	TCP_NODELAY	int	0

Protocol Level	Option Name	Option data type	Option value
			or 1

Returns

Value Meaning

0	Successful set of option
-1	An error occurred

getsockopt will fail if:

EBADF	The socket descriptor is invalid
EINVAL	One of the parameters is invalid
ENOPROTOOPT	The option is unknown at the level indicated.

get_status

Get Serial Port Status

Syntax

```
#include <ctools.h>
struct pstatus *get_status(UCHAR port, struct pstatus *status);
```

Description

The **get_status** function returns serial port error counters, I/O lines status and I/O driver buffer information for *stream*. If *port* is not a valid serial port the function has no effect. *status* must point to a valid serial port status structure, *pstatus*.

The **get_status** function copies the serial port status into the structure pointed to by *status* and returns a pointer to the structure *settings*.

Refer to the **Overview of Functions** section for detailed information on serial ports.

See Also

clear_errors

Example

This program displays the framing and parity errors for **com1**.

```
#include <ctools.h>

int main(void)
{
       struct pstatus status;

       get_status(com1, &status);
       fprintf(com1,"Framing: %d\r\n", status.framing);
       fprintf(com1,"Parity:  %d\r\n", status.parity);
}
```

getStatusBit

Read Bits in Controller Status Code

Syntax

```
#include <ctools.h>
UINT16 getStatusBit(UINT16 bitMask);
```

Description

The **getStatusBit** function returns the values of the bits indicated by *bitMask* in the controller status code.

See Also

setStatusBit**,** clearStatusBit

getTaskInfo

Get Information on a Task

Syntax

```
#include <ctools.h>
BOOLEAN getTaskInfo(UINT32 taskID, TASKINFO *pTaskInfo);
```

Description

The **getTaskInfo** function returns information about the task specified by *taskID*. If *taskID* is 0 the function returns information about the current task. The function copies task information to the TASKINFO structure pointed to by *pTaskInfo*.

FALSE is returned if the task specified by *taskID* doesn't exist; otherwise TRUE is returned and the data is copied.

Refer to the *Structures and Types* section for a description of the fields in the TASKINFO structure.

Example

See the Get Task Status Example in the Examples section.

getVersion

Get Firmware Version Information

Syntax

```
#include <ctools.h>
VERSION getVersion(void);
```

Description

The **getVersion** function obtains firmware version information. It returns a VERSION structure. Refer to the **Structures and Types** section for a description of the fields in the VERSION structure.

Notes

The version information can be used to adapt a program to a specific type of controller or version of firmware. For example, a bug work-around could be executed only if older firmware is detected.

Example

This program displays the version information.

```
#include <ctools.h>
int main(void)
{
      struct prot_settings settings;
      VERSION versionInfo;

      /* Disable the protocol on serial port 1 */
      settings.type =        NO_PROTOCOL;
      settings.station =     1;
      settings.priority =    250;
      settings.SFMessaging = FALSE;
      request_resource(IO_SYSTEM);
      set_protocol(com1, &settings);
      release_resource(IO_SYSTEM);

      /* Display the ROM version information */
      versionInfo = getVersion();
      fprintf(com1, "\r\nFirmware Information\r\n");

      fprintf(com1, "  Controller type:  %d\r\n",
         versionInfo.controller);
      fprintf(com1, "  Firmware version: %d\r\n",
         versionInfo.version);
      fprintf(com1, "  Creation date:    %s\r\n",
         versionInfo.date);
      fprintf(com1, "  Copyright:        %s\r\n",
         versionInfo.copyright);
}
```

getWakeSource

Gets Conditions for Waking from Sleep Mode

Syntax

```
#include <ctools.h>
UINT32 getWakeSource(void);
```

Description

The **getWakeSource** function returns a bit mask of the active wake up sources. Valid wake up sources are listed below.

- WAKE_SOURCE_RTC_ALARM
- WAKE_SOURCE_COUNTER_1_OVERFLOW
- WAKE_SOURCE_COUNTER_2_OVERFLOW
- WAKE_SOURCE_COUNTER_3_OVERFLOW
- WAKE_SOURCE_LED_POWER_SWITCH
- WAKE_SOURCE_DIN_1_CHANGE
- WAKE_SOURCE_COM3_VISION

See Also

setPowerMode

Example

The following code fragment displays the enabled wake up sources.

```
unsigned enabled;

enabled = getWakeSource();
fputs("Enabled wake up sources:\r\n", com1);
if (enabled & WAKE_SOURCE_RTC_ALARM)
   fputs("  Real Time Clock\r\n", com1);
if (enabled & WAKE_SOURCE_LED_POWER_SWITCH)
   fputs("  LED Power Switch\r\n", com1);
if (enabled & WAKE_SOURCE_COUNTER_1_OVERFLOW)
   fputs("  Counter 1 Overflow\r\n", com1);
if (enabled & WAKE_SOURCE_COUNTER_1_OVERFLOW)
   fputs("  Counter 2 Overflow\r\n", com1);
if (enabled & WAKE_SOURCE_COUNTER_1_OVERFLOW)
   fputs("  Counter 3 Overflow\r\n", com1);
```

Handler Function

User Specified Handler Function

The handler function is a user-specified function that handles processing of Modbus messages not recognized by the protocol. The function can have any name; *handler* is used in the description below.

Syntax

```
#include <ctools.h>
UINT16 handler(
       UCHAR * message,
       UINT16 messageLength,
       UCHAR * response,
       UINT16 * responseLength
       );
```

Description

This function *handler* is a user-defined handler for processing Modbus messages. The function is called for each Modbus message with a function code that is not recognized by the standard Modbus protocol.

The *handler* function should process the message string and create a response string. If the message is not understood, one of the error codes should be returned.

The function has four parameters.

- The *message* parameter is a pointer to the first character of the received message. The first character of the message is the function code. The format of the data after the function code is defined by the function code.
- The *messageLength* parameter is the number of characters in the message.
- The *response* parameter is a pointer to the first character of a buffer to hold the response. The function should write the response into this buffer. The buffer is 253 characters long. The first character of the buffer is the function code of the message. The format of the data after the function code is defined by the function code.
- The *responseLength* parameter is a pointer to the length of the response. The function should set the length of the response using this pointer. The length is the number of characters placed into the response buffer.

The function must return one of four values. The first causes a normal response to be sent. The others cause an exception response to be sent.

- NORMAL indicates the response and responseLength have been set to valid values. The Modbus protocol will add the station address and checksum to this string and transmit the reply to the master station.
- ILLEGAL_FUNCTION indicates the function code in the message was understood, but the function was deemed illegal.
- ILLEGAL_DATA_ADDRESS indicates the function code in the message was understood, but that the command referenced an address that is not valid. The Modbus protocol will return an Illegal Data Address exception response.
- ILLEGAL_DATA_VALUE indicates the function code in the message was understood, but that the command included data that is not valid. The Modbus protocol will return an Illegal Data Address exception response.

- FUNCTION_NOT_HANDLED must be returned by the function handler if the function was not handled. If no installed handler can process the function then an ILLEGAL_FUNCTION exception response will be sent.

Function Codes Used

The following function codes are currently used by the TeleBUS Modbus-compatible protocol. All other function codes are available for use. For maximum compatibility with other Modbus and Modbus-compatible devices it is recommended that codes in the user-defined function code range be used first.

Code	Type	Description
1	Modbus standard	Read coil registers from I/O database
2	Modbus standard	Read status registers from I/O database
3	Modbus standard	Read holding registers from I/O database
4	Modbus standard	Read input registers from I/O database
5	Modbus standard	Write a single coil register
6	Modbus standard	Write a single holding register
7	Modbus standard	Read exception status
15	Modbus standard	Write multiple coil registers
16	Modbus standard	Write multiple holding registers
17	Modbus standard	Report slave identification string
65	TeleBUS extension	Used by TelePACE
66	TeleBUS extension	Used by TelePACE
67	TeleBUS extension	Used by TelePACE
68	TeleBUS extension	Used by TelePACE
69	TeleBUS extension	Used by TelePACE
70	TeleBUS extension	Used by TelePACE
71	TeleBUS extension	Used by TelePACE

Notes

One *handler* function is used for all serial ports. Only one port will be active at any time. Therefore, the function does not have to be re-entrant.

The *handler* function is called from the Modbus protocol task. This task may pre-empt the execution of another task. If there are shared resources, the *handler* function must request and release the appropriate resources to ensure proper operation.

The station address is not included in the message or response string. It will be added to the response string before sending the reply.

The checksum is not included in the message or the response string. It will be added to the response string before sending the reply.

The maximum size of the response string is 253 bytes. If a longer response length is returned, the Modbus protocol will report an ILLEGAL_DATA_VALUE exception. The response will not be returned.

See Also

installModbusHandler

hartIO

Read and Write 5904 HART Interface Module

Syntax

```
#include <ctools.h>
BOOLEAN hartIO(UINT16 module)
```

Description

This function reads the specified 5904 interface module. It checks if a response has been received and if a corresponding command has been sent. If so, the response to the command is processed.

This function writes the specified 5904 interface module. It checks if there is a new command to send. If so, this command is written to the 5904 interface.

The function has one parameter: the module number of the 5904 interface (0 to 3).

The I/O read and write operations are added to the I/O System queue.

The function returns TRUE if the 5904 interface responded to the previous I/O request and FALSE if it did not or if the module number is not valid.

Notification of the completion of I/O requests made by this function may be obtained using the **ioNotification** function.

See Also

hartSetConfiguration, hartGetConfiguration, hartCommand, ioNotification

hartCommand

Send Command using HART Interface Module

Syntax

```
#include <ctools.h>
BOOLEAN hartCommand(
      UINT16 module,
      HART_DEVICE * const device,
      HART_COMMAND * const command,
      void (* processResponse)( UINT16,
      HART_RESPONSE)
      );
```

Description

This function sends a command to a HART slave device using a HART interface module. This function can be used to implement HART commands not provided by the Network Layer API.

The function has four parameters. The first is the module number of the 5904 interface (0 to 3). The second is the device to which the command is to be sent.

The third parameter is a structure describing the command to send. This contains the command number, and the data field of the HART message. See the HART protocol documentation for your device for details.

The fourth parameter is a pointer to a function that will process the response. This function is called when a response to the command is received by the HART interface. The function is defined as follows:

```
void function_name(HART_RESPONSE response)
```

The single parameter is a structure containing the response code and the data field from the message.

The function returns TRUE if the 5904 interface responded and FALSE if it did not or if the module number is not valid or there is an error in the command.

Notes

The function returns immediately after the command is sent. The calling program must wait for the response to be received. Use the hartStatus command to read the status of the command.

The number of attempts and the number of preambles sent are set with the hartSetConfiguration command.

A program must initialize the link before executing any other commands.

The function determines if long or short addressing is to be used by the command number. Long addressing is used for all commands except commands 0 and 11.

The functions hartCommand0, hartCommand1, etc. are used to send commands provided by the Network Layer.

See Also

hartStatus**,** hartSetConfiguration**,** hartCommand0**,** hartCommand1

hartCommand0

Read Unique Identifier

Syntax

```
#include <ctools.h>
BOOLEAN hartCommand0(UINT16 module, UINT16 address, HART_DEVICE * const
device);
```

Description

This function reads the unique identifier of a HART device using command 0 with a short-form address. This is a link initialization function.

The function has three parameters: the module-number of the 5904 module (0 to 3); the short-form address of the HART device (0 to 15); and a pointer to a HART_DEVICE structure. The information read by command 0 is written into the HART_DEVICE structure when the response is received by the 5904 interface.

The function returns TRUE if the command was sent. The function returns FALSE if the module number is invalid, or if the device address is invalid.

Notes

The function returns immediately after the command is sent. The calling program must wait for the response to be received. Use the hartStatus command to read the status of the command.

The number of attempts and the number of preambles sent are set with the hartSetConfiguration command.

A program must initialize the link before executing any other commands.

See Also

hartCommand11**,** hartStatus**,** hartSetConfiguration

hartCommand1

Read Primary Variable

Syntax

```
#include <ctools.h>
BOOLEAN hartCommand1(UINT16 module, HART_DEVICE * const device,
HART_VARIABLE * primaryVariable);
```

Description

This function reads the primary variable of a HART device using command 1.

The function has three parameters: the module-number of the 5904 module (0 to 3); the device to be read; and a pointer to the primary variable. The variable pointed to by primaryVariable is updated when the response is received by the 5904 interface.

The primaryVariable must be a static modular or global variable. A primaryVariable should be declared for each HART I/O module in use. A local variable or dynamically allocated variable may not be used because a late command response received after the variable is freed will write data over the freed variable space.

The function returns TRUE if the command was sent. The function returns FALSE if the module number is invalid.

Notes

The HART_DEVICE structure must be initialized using hartCommand0 or hartCommand11.

The function returns immediately after the command is sent. The calling program must wait for the response to be received. Use the hartStatus command to read the status of the command.

The number of attempts and the number of preambles sent are set with the hartSetConfiguration command.

The code field of the HART_VARIABLE structure not changed. Command 1 does not return a variable code.

See Also

hartCommand2**,** hartStatus**,** hartSetConfiguration

hartCommand2

Read Primary Variable Current and Percent of Range

Syntax

```
#include <ctools.h>
BOOLEAN hartCommand2(UINT16 module, HART_DEVICE * const device,
HART_VARIABLE * pvCurrent, HART_VARIABLE * pvPercent);
```

Description

This function reads the primary variable (PV), as current and percent of range, of a HART device using command 2.

The function has four parameters: the module-number of the 5904 module (0 to 3); the device to be read; a pointer to the PV current variable; and a pointer to the PV percent variable. The pvCurrent and pvPercent variables are updated when the response is received by the 5904 interface.

The pvCurrent and pvPercent variables must be static modular or global variables. A pvCurrent and pvPercent variable should be declared for each HART I/O module in use. A local variable or dynamically allocated variable may not be used because a late command response received after the variable is freed will write data over the freed variable space.

The function returns TRUE if the command was sent. The function returns FALSE if the module number is invalid.

Notes

The HART_DEVICE structure must be initialized using hartCommand0 or hartCommand11.

The function returns immediately after the command is sent. The calling program must wait for the response to be received. Use the hartStatus command to read the status of the command.

The number of attempts and the number of preambles sent are set with the hartSetConfiguration command.

The code field of both HART_VARIABLE structures is not changed. The response from the HART device to command 2 does not include variable codes.

The units field of the pvCurrent variable is set to 39 (units = mA). The units field of the pvPercent variable is set to 57 (units = percent). The response from the HART device to command 2 does not include units.

See Also

hartCommand1**,** hartStatus**,** hartSetConfiguration

hartCommand3

Read Primary Variable Current and Dynamic Variables

Syntax

```
#include <ctools.h>
BOOLEAN hartCommand3(UINT16 module, HART_DEVICE * const device,
HART_VARIABLE * variables);
```

Description

This function reads dynamic variables and primary variable current from a HART device using command 3.

The function has three parameters: the module number of the 5904 module (0 to 3); the device to be read; and a pointer to an array of five HART_VARIABLE structures.

The variables array must be static modular or global variables. An array of variables should be declared for each HART I/O module in use. A local variable or dynamically allocated variable may not be used because a late command response received after the variable is freed will write data over the freed variable space.

The variables array is updated when the response is received by the 5904 interface as follows.

Variable	Contains
Variables[0]	primary variable current
Variables[1]	primary variable
Variables[2]	secondary variable
Variables[3]	tertiary variable
Variables[4]	fourth variable

The function returns TRUE if the command was sent. The function returns FALSE if the module number is invalid.

Notes

The HART_DEVICE structure must be initialized using hartCommand0 or hartCommand11.

The function returns immediately after the command is sent. The calling program must wait for the response to be received. Use the hartStatus command to read the status of the command.

The number of attempts and the number of preambles sent are set with the hartSetConfiguration command.

Not all devices return primary, secondary, tertiary and fourth variables. If the device does not support a variable, zero is written into the value and units code for that variable.

The code field of both HART_VARIABLE structures is not changed. The response from the HART device to command 3 does not include variable codes.

The units field of variable[0] is set to 39 (units = mA). The response from the HART device to command 3 does not include units.

See Also

hartCommand33, hartStatus, hartSetConfiguration

hartCommand11

Read Unique Identifier Associated with Tag

Syntax

```
#include <ctools.h>
BOOLEAN hartCommand11(UINT16 module, char * deviceTag, HART_DEVICE *
device);
```

Description

This function reads the unique identifier of a HART device using command 11. This is a link initialization function.

The function has three parameters: the module number of the 5904 module (0 to 3); a pointer to a null terminated string containing the tag of the HART device; and a pointer to a HART_DEVICE structure. The information read by command 11 is written into the HART_DEVICE structure when the response is received by the 5904 interface.

The function returns TRUE if the command was sent. The function returns FALSE if the module number is invalid.

Notes

The function returns immediately after the command is sent. The calling program must wait for the response to be received. Use the hartStatus command to read the status of the command.

The number of attempts and the number of preambles sent are set with the hartSetConfiguration command.

A program must initialize the link before executing any other commands.

See Also

hartCommand0**,** hartStatus**,** hartSetConfiguration

hartCommand33

Read Transmitter Variables

Syntax

```
#include <ctools.h>
BOOLEAN hartCommand33(UINT16 module, HART_DEVICE * const device, UINT16
variableCode[4], HART_VARIABLE * variables);
```

Description

This function reads selected variables from a HART device using command 33.

The function has four parameters: the module number of the 5904 module (0 to 3); the device to be read; an array of codes; and a pointer to an array of four HART_VARIABLE structures.

The variables array must be static modular or global variables. An array of variables should be declared for each HART I/O module in use. A local variable or dynamically allocated variable may not be used because a late command response received after the variable is freed will write data over the freed variable space.

The variableCode array specifies which variables are to be read from the transmitter. Consult the documentation for the transmitter for valid values.

The variables array is updated when the response is received by the 5904 interface as follows.

Variable	Contains
Variables[0]	transmitter variable, code and units specified by variableCode[0]
Variables[1]	transmitter variable, code and units specified by variableCode[1]
Variables[2]	transmitter variable, code and units specified by variableCode[2]
Variables[3]	transmitter variable, code and units specified by variableCode[3]

The function returns TRUE if the command was sent. The function returns FALSE if the module number is invalid.

Notes

The HART_DEVICE structure must be initialized using hartCommand0 or hartCommand11.

The pointer variables must point to an array with at least four elements.

The function returns immediately after the command is sent. The calling program must wait for the response to be received. Use the hartStatus command to read the status of the command.

The number of attempts and the number of preambles sent are set with the hartSetConfiguration command.

The function always requests four variables and expects four variables in the response.

See Also

hartCommand3**,** hartStatus**,** hartSetConfiguration

hartStatus

Return Status of Last HART Command Sent

Syntax

```
#include <ctools.h>
BOOLEAN hartStatus(UINT16 module, HART_RESULT * status, UINT16 * code);
```

Description

This function returns the status of the last HART command sent by a 5904 module (0 to 3). Use this function to determine if a response has been received to a command sent.

The function has three parameters: the module number of the 5904 module; a pointer to the status variable; and a pointer to the additional status code variable. The status and code variables are updated with the following information.

Result	Status	code
HART interface module is not communicating	HR_NoModuleResponse	not used
Command ready to be sent	HR_CommandPending	not used
Command sent to device	HR_CommandSent	current attempt number
Response received	HR_Response	response code from HART device (see Notes)
No valid response received after all attempts made	HR_NoResponse	0=no response from HART device. Other = error response code from HART device (see Notes)
HART interface module is not ready to transmit	HR_WaitTransmit	not used

The function returns TRUE if the status was read. The function returns FALSE if the module number is invalid.

Notes

The response code from the HART device contains communication error and status information. The information varies by device, but there are some common values.

- If bit 7 of the high byte is set, the high byte contains a communication error summary. This field is bit-mapped. The table shows the meaning of each bit as defined by the HART protocol specifications. Consult the documentation for the HART device for more information.

Bit	Description
6	vertical parity error
5	overrun error
4	framing error
3	longitudinal parity error
2	reserved – always 0
1	buffer overflow
0	Undefined

- If bit 7 of the high byte is cleared, the high byte contains a command response summary. The table shows common values. Other values may be defined for specific commands. Consult the documentation for the HART device.

Code	Description
32	Busy – the device is performing a function that cannot be interrupted by this command
64	Command not Implemented – the command is not defined for this device.

- The low byte contains the field device status. This field is bit-mapped. The table shows the meaning of each bit as defined by the HART protocol specifications. Consult the documentation for the HART device for more information.

Bit	Description
7	field device malfunction
6	configuration changed
5	cold start
4	more status available (use command 48 to read)
3	primary variable analog output fixed
2	primary variable analog output saturated
1	non-primary variable out of limits
0	primary variable out of limits

See Also

hartSetConfiguration

hartGetConfiguration

Read HART Module Settings

Syntax

```
#include <ctools.h>
BOOLEAN hartGetConfiguration(UINT16 module, HART_SETTINGS * settings);
```

Description

This function returns the configuration settings of a 5904 module.

The function has two parameters: the module number of the 5904 module (0 to 3); and a pointer to the settings structure.

The function returns TRUE if the settings were read. The function returns FALSE if the module number is invalid.

See Also

hartSetConfiguration

hartSetConfiguration

Write HART Module Settings

Syntax

```
#include <ctools.h>
BOOLEAN hartSetConfiguration(UINT16 module, HART_SETTINGS settings);
```

Description

This function writes configuration settings to a 5904 module.

The function has two parameters: the module number of the 5904 module (0 to 3); and a settings structure.

The function returns TRUE if the settings were written. The function returns FALSE if the module number or the settings are invalid.

Notes

The configuration settings are stored in flash. The user-defined settings are used when the controller is reset in the RUN mode. Default settings are used when the controller is reset in the SERVICE or COLD BOOT modes. To save these settings with the controller settings in flash memory so that they are loaded on controller reset, call **flashSettingsSave** as shown below.

```
request_resource(FLASH_MEMORY);
flashSettingsSave(CS_RUN);
release_resource(FLASH_MEMORY);
```

See Also

hartGetConfiguration

hartPackString

Convert String to HART Packed String

Syntax

```
#include <ctools.h>
void hartPackString(CHAR * pPackedString, const CHAR * pString, UINT16
sizePackedString);
```

Description

This function stores an ASCII string into a HART packed ASCII string.

The function has three parameters: a pointer to a packed array; a pointer to an unpacked array; and the size of the packed array. The packed array must be a multiple of three in size. The unpacked array must be a multiple of four in size. It should be padded with spaces at the end if the string is not long enough.

The function has no return value.

See Also

hartUnpackString

hartUnpackString

Convert HART Packed String to String

Syntax

```
#include <ctools.h>
void hartUnpackString(CHAR * pString, const CHAR * pPackedString, UINT16
sizePackedString);
```

Description

This function unpacks a HART packed ASCII string into a normal ASCII string.

The function has three parameters: a pointer to an unpacked array; a pointer to a packed array; and the size of the packed array. The packed array must be a multiple of three in size. The unpacked array must be a multiple of four in size.

The function has no return value.

See Also

hartPackString

htonl

Syntax

```
#include <ctools.h>
unsigned long htonl
(
       unsigned long longValue
);
```

Function Description

This function converts a long value from host byte order to network byte order.

Parameters

longValue	The value to convert

Returns

The converted value.

htons

Syntax

```
#include <ctools.h>
unsigned short htons
(
      unsigned short shortValue
);
```

Function Description

This function converts a short value from host byte order to network byte order.

Parameters

shortValue	The value to convert

Returns

The converted value.

inet_addr

Syntax

```
#include <ctools.h>
unsigned long inet_addr
(
       char * ipAddressDottedStringPtr
);
```

Function Description

This function converts an IP address from the decimal dotted notation to an unsigned long.

Parameters

ipAddressDottedStringPtr The dotted string (i.e. “208.229.201.4”)

Returns

Value	Meaning
-1	Error
Other	The IP Address in Network Byte Order.

install_handler

Install Serial Port Handler

Syntax

```
#include <ctools.h>
void install_handler(FILE *stream, void *function(UINT, UINT));
```

Description

The **install_handler** function installs a serial port character handler function. The serial port driver calls this function each time it receives a character. If *stream* does not point to a valid serial port the function has no effect.

function specifies the handler function, which takes two arguments. The first argument is the received character. The second argument is an error flag. A non-zero value indicates an error. If *function* is **NULL**, the default handler for the port is installed. The default handler does nothing.

Notes

The **install_handler** function can be used to write custom communication protocols.

The handler is called at the completion of the receiver interrupt handler. RTOS calls (see functions listed in the section *Real Time Operating System Functions* at the start of this chapter) may not be made within the interrupt handler, with one exception. The **interrupt_signal_event** RTOS call can be used to signal events.

To optimize performance, minimize the length of messages on com3. Examples of recommended uses for com3 are for local operator display terminals, and for programming and diagnostics using the ISaGRAF program.

Example

See the Install Serial Port Handler Example in the Examples section.

installClockHandler

Install Handler for Real Time Clock

Syntax

```
#include <ctools.h>
void installClockHandler(void (*function)(void));
```

Description

The **installClockHandler** function installs a real time clock alarm handler function. The real time clock alarm function calls this function each time a real time clock alarm occurs.

function specifies the handler function. If *function* is **NULL**, the handler is disabled.

Notes

RTOS calls (see functions listed in the section *Real Time Operating System Functions* at the start of this chapter) may not be made within the interrupt handler, with one exception. The **interrupt_signal_event** RTOS call can be used to signal events.

See Also

setClockAlarm

Example

See the Install Clock Handler Example in the Examples section.

installDbaseHandler

Install User Defined Dbase Handler (ISaGRAF firmware only)

Syntax

```
#include <ctools.h>
void installDbaseHandler
   (
   BOOLEAN (* handler)
      (
      UINT16 address,
      int  *value
      )
   )
```

Description

The **installDbaseHandler** function allows an extension to be defined for the dbase() function.

If a handler is installed, it is called by the dbase function when one of the following conditions apply:

- There is no ISaGRAF application downloaded, or
- There is no ISaGRAF variable assigned to the specified Modbus address.

The function installDbaseHandler has one parameter: a pointer to a function to handle the dbase extensions. See the section *Dbase Handler Function* for a full description of the handler function and it's parameters. If the pointer is NULL, no handler is installed.

The installed handler is always called with a Modbus address. Linear addresses are converted to Modbus addresses before calling the handler. Use the **installSetdbaseHandler** function to install a write access handler for the same addresses handled by the dbase handler.

Note that the C++ Tools functions dbase and setdbase are used by all protocols to access Modbus or Linear registers.

Notes

Call this function with the NULL pointer to remove the dbase handler. This must be done when the application program is ended with an exit handler. Use the installExitHandler function to install the exit handler.

If the Dbase handler is not removed within an exit handler, it will remain installed and continue to operate until the controller power is cycled. *Erasing the C Program* from the Initialize dialog will not remove the Dbase handler. If the handler is located in a RAM-based application and left installed while a different C application is downloaded, the original handler will be corrupted and the system will likely crash.

installSetdbaseHandler

Install User Defined Setdbase Handler (ISaGRAF firmware only)

Syntax

```
#include <ctools.h>
void installSetdbaseHandler
   (
   BOOLEAN (* handler)
      (
      UINT16 address,
      int  value
      )
   )
```

Description

The **installSetdbaseHandler** function allows an extension to be defined for the setdbase() function.

If a handler is installed, it is called by the setdbase function when one of the following conditions apply:

- There is no ISaGRAF application downloaded, or
- There is no ISaGRAF variable assigned to the specified Modbus address.

The function installSetdbaseHandler has one parameter: a pointer to a function to handle the setdbase extensions. See the section *Setdbase Handler Function* for a full description of the handler function and it's parameters. If the pointer is NULL, no handler is installed.

The installed handler is always called with a Modbus address. Linear addresses are converted to Modbus addresses before calling the handler. Use the **installDbaseHandler** function to install a read access handler for the same addresses handled by the setdbase handler.

Note that the C++ Tools functions dbase and setdbase are used by all protocols to access Modbus or Linear registers.

Notes

Call this function with the NULL pointer to remove the setdbase handler. This must be done when the application program is ended with an exit handler. Use the installExitHandler function to install the exit handler.

If the Setdbase handler is not removed within an exit handler, it will remain installed and continue to operate until the controller power is cycled. *Erasing the C Program* from the Initialize dialog will not remove the Setdbase handler. If the handler is located in a RAM-based application and left installed while a different C application is downloaded, the original handler will be corrupted and the system will likely crash.

See Also

setdbase**,** installDbaseHandler

Example

See example for Setdbase Handler Function.

installExitHandler

Install Handler Called when Task Ends

Syntax

```
#include <ctools.h>
BOOLEAN installExitHandler(UINT32 taskID, FUNCPTR function) );
```

Description

The **installExitHandler** function defines a function that is called when the task, specified by *taskID*, is ended. *function* specifies the handler function. If *function* is **NULL**, the handler is disabled.

Notes

The exit handler function will be called when:

- the task is ended by the end_task or end_group function
- the end_application function is executed and the function is an APPLICATION type function
- the program is stopped from the ISaGRAF or TelePACE program and the task is an APPLICATION type function
- the program is erased by the ISaGRAF or TelePACE program.

The exit handler function is not called if power to the controller is removed. In this case all execution stops when power fails. The application program starts from the beginning when power is reapplied.

Do not call any RTOS functions from the exit handler.

Example

See the example for startTimedEvent.

installModbusHandler

Install User Defined Modbus Handler

Syntax

```
#include <ctools.h>
void installModbusHandler(
UINT16 (* handler)(UCHAR *, UINT16,
                   UCHAR *, UINT16 *)
      );
```

Description

The installModbusHandler function allows user-defined extensions to standard Modbus protocol. This function specifies a function to be called when a Modbus message is received for the station, but is not understood by the standard Modbus protocol. The installed handler function(s) is called only if the message is addressed to the station, and the message checksum is correct.

The function has one parameter: a pointer to a function to handle the messages. See the section **Handler Function** for a full description of the function and it's parameters. The function has no return value.

Notes

This function is used to create a user-defined extension to the standard Modbus protocol.

Call the **removeModbusHandler** function to remove a previously installed handler. This must be done when the application program is ended with an exit handler. Use the installExitHandler function to install the exit handler.

If the Modbus handler is not removed within an exit handler, it will remain installed and continue to operate until the controller power is cycled. Changing the protocol type or *Erasing the C Program* from ISaGRAF Initialize dialog will not remove the Modbus handler. If the handler is located in a RAM-based application and left enabled while a different C application is downloaded, the original handler will be corrupted and the system will likely crash.

See Also

removeModbusHandler, Handler Function, installExitHandler

installRTCHandler

Install User Defined Real-Time-Clock Handler

Syntax

```
#include <ctools.h>
void installRTCHandler(
      void (* rtchandler)(TIME *now, TIME *new));
```

Description

The installRTCHandler function allows an application program to override Modbus protocol and DNP protocol commands to set the real time clock. This function specifies a function to be called when a Modbus or DNP message is received for the station. The installed handler function is called only if the message is intended to set the real time clock.

The function has one parameter: a pointer to a function to handle the messages. See the section **RTCHandler Function** for a full description of the function and its parameters. If the pointer is NULL, no function is called for set the real time clock commands, and the default method is used set the real time clock.

The function has no return value.

Notes

Call this function with the NULL pointer to disable processing of *Set Real Time Clock* messages. This must be done when the application program is ended with an exit handler. Use the installExitHandler function to install the exit handler.

If the RTC handler is not disabled within an exit handler, it will remain installed and continue to operate until the controller power is cycled. Changing the protocol type or *Erasing the C Program* from the TelePACE Initialize dialog will not remove the handler. If the handler is located in a RAM-based application and left enabled while a different C application is downloaded, the original handler will be corrupted and the system will likely crash.

See Also

RTCHandler Function, installExitHandler

RTCHandler Function

User Specified Real Time Clock Handler Function

The handler function is a user-specified function that handles processing of Modbus messages or DNP messages for setting the real time clock. The function can have any name; *rtchandler* is used in the description below.

Syntax

```
#include <ctools.h>
void rtchandler(
      TIME *now,
      TIME *new
      );
```

Description

This function *rtchandler* is a user-defined handler for processing Modbus messages or DNP messages. The function is called only for messages that set the real time clock.

The *rtchandler* function should set the real time clock to the requested time. If there is a delay before this can be done, the time when the message was received is provided so that a correction to the requested time can be made.

The function has two parameters.

- The *now* parameter is a pointer to the structure containing the time when the message was received.
- The *new* parameter is a pointer to the structure containing the requested time.

The function does not return a value.

Notes

The IO_SYSTEM resource has already been requested before calling this function. If this function calls other functions that require the IO_SYSTEM resource (e.g. setclock), there is no need to request or release the resource.

This function must not request or release the IO_SYSTEM resource.

See Also

installRTCHandler

ioClear

Turn Off all Outputs

Syntax

```
#include <ctools.h>
void ioClear(void)
```

Description

The **ioClear** function turns off all outputs as follows.

- a reset of all I/O modules is added to the I/O System queue;
- analog outputs are set to 0;
- digital outputs are set to 0 (turned off).

Notification of the completion of I/O requests made by this function may be obtained using the **ioNotification** function.

Notes

The IO_SYSTEM resource must be requested before calling this function.

ioDatabaseReset

Initialize I/O Database with Default Values

Syntax

```
#include <ctools.h>
void ioDatabaseReset(void);
```

Description

The **ioDatabaseReset** function resets the target controller to default settings.

- Configuration parameters are reset to the default values.
- Communication status counters are reset to zero.
- Output I/O points are cleared.
- Locked variables are unlocked.
- Clear all I/O forcing
- Clear all I/O points
- Set all database locations to zero
- Set I/O database for real-time clock to current time
- Clear real time clock alarm settings
- Configure serial ports with default parameters
- Configure serial ports with default protocols
- Clear serial port event counters
- Clear store and forward configuration
- Enable LED power by default and return to default state after 5 minutes
- Set Outputs on Stop settings to Hold
- Set 5904 HART modem configuration for all modems
- Set Modbus/TCP default configuration
- Write new default data to Flash

Notes

This function can be used to restore the controller to its default state. **ioDatabaseReset** has the same effect as selecting the **Initialize Controller** option from the **Initialize** command in the ISaGRAF program.

The IO_SYSTEM resource must be requested before calling this function.

Example

```
#include <ctools.h>

int main(void)
{
       /* Power Up Initialization */
       request_resource(IO_SYSTEM);
       ioDatabaseReset();
       release_resource(IO_SYSTEM);

       /* ... the rest of the program */
}
```

ioGetConfiguration

Get I/O Controller Configuration

Syntax

```
#include <ctools.h>
IO_CONFIG& ioGetConfiguration(void)
```

Description

This function returns the I/O controller configuration.

The function has no arguments.

The function returns an IO_CONFIG structure containing the configuration.

ioNotification

Add I/O Notification Request

Syntax

```
#include <ctools.h>
BOOLEAN ioNotification(UINT16 eventNumber)
```

Description

This function adds a Notification Request to the I/O Controller request queue. The specified event number is signaled when the notification request is processed.

The function has one argument: an event number. Valid events are numbered 0 to 31.

The function returns TRUE if the request was added. The function returns FALSE if there is no room in the request queue or if the event number is invalid.

ioRead5505Inputs

Read 5505 Inputs

Syntax

```
#include <ctools.h>
BOOLEAN ioRead5505Inputs(
      UINT16 moduleAddress,
      UINT16 &dinData,
      float (&ainData)[4]
)
```

Description

This function reads buffered data from the digital and analog inputs of a 5505 I/O module. Buffered data are updated when an I/O request for the module is processed.

`moduleAddress` is the address of the 5505 module. Valid values are 0 to 15.

`dinData` is a reference to a UINT16 variable. Digital data for the 16 internal inputs are written to this variable. One bit in the variable represents each input point. The function of the 16 digital inputs is described in the table below.

Point Offset	Function
0	OFF = channel 0 RTD is good ON = channel 0 RTD is open or PWR input is off
1	OFF = channel 0 data in range ON = channel 0 data is out of range
2	OFF = channel 0 RTD is using 3-wire measurement ON = channel 0 RTD is using 4-wire measurement
3	reserved for future use
4	OFF = channel 1 RTD is good ON = channel 1 RTD is open or PWR input is off
5	OFF = channel 1 data in range ON = channel 1 data is out of range
6	OFF = channel 1 RTD is using 3-wire measurement ON = channel 1 RTD is using 4-wire measurement
7	reserved for future use
8	OFF = channel 2 RTD is good ON = channel 2 RTD is open or PWR input is off
9	OFF = channel 2 data in range ON = channel 2 data is out of range
10	OFF = channel 2 RTD is using 3-wire measurement ON = channel 2 RTD is using 4-wire measurement
11	reserved for future use
12	OFF = channel 3 RTD is good ON = channel 3 RTD is open or PWR input is off
13	OFF = channel 3 data in range ON = channel 3 data is out of range
14	OFF = channel 3 RTD is using 3-wire measurement ON = channel 3 RTD is using 4-wire measurement
15	Reserved for future use

`ainData` is a reference to an array of four floating point variables. Analog data are written to this array.

The function returns FALSE if the module address is invalid; otherwise TRUE is returned.

See Also

ioRead5505Outputs, **ioWrite5505Outputs**

Example

This program displays the values of the 16 internal digital inputs and the 4th analog input read from 5505 I/O at address 5.

```
#include <ctools.h>
#define     MY_EVENT    1

int main(void)
{
      UINT16      dinData;
      float       ainData[4];
      IO_STATUS   io_status;
      BOOLEAN     status;
      BOOLEAN     done;

      // main loop
      while (TRUE)
      {
            // add module scan to queue
            if (!ioRequest(MT_5505Inputs, 5))
            {
                  status = FALSE;
            }

            // wait for scan to complete
            ioNotification(MY_EVENT);
            wait_event(MY_EVENT);

            // read input data from last scan
            status = ioRead5505Inputs(5, dinData, ainData);

            // check status of last scan
            if (!ioStatus(MT_5505Inputs, 5, &io_status))
            {
                  status = FALSE;
            }
            else if (!io_status.commStatus)
            {
                  status = FALSE;
            }

            // print data when option switch 4 is selected
            if (optionSwitch(4))
            {
                  if (!done)
                  {
```

```
                        fprintf(com1, "status = %u,\
                        Dins 0 to 15 = %X, Ain 3 = %f\r\n",
                        status, dinData, ainData[3]);
                        done = TRUE;
                  }
            }
            else
            {
                  done = FALSE;
            }

            // release processor to other priority 1 tasks
            release_processor();
      }
}
```

ioRead5505Outputs

Read 5505 Configuration

Syntax

```
#include <ctools.h>
BOOLEAN ioRead5505Outputs(
      UINT16 moduleAddress,
      UINT16 (&inputType)[4],
      UINT16 &inputFilter
)
```

Description

This function reads configuration data from the I/O Table for a 5505 I/O module. Configuration data are written using the `ioWrite5505Outputs` function.

`moduleAddress` is the address of the 5505 module. Valid values are 0 to 15.

`inputType` is a reference to an array of four UINT16 variables. Analog input measurement types are written to this array. Valid values are

- 0 = RTD in deg Celsius
- 1 = RTD in deg Fahrenheit
- 2 = RTD in deg Kelvin
- 3 = resistance measurement in ohms.

`inputFilter` is a reference to a UINT16 variable. The input filter selection is written to this variable.

- 0 = 0.5 s
- 1 = 1 s
- 2 = 2 s
- 3 = 4 s

The function returns FALSE if the module address is invalid; otherwise TRUE is returned.

See Also

ioRead5505Inputs, **ioWrite5505Outputs**

Example

This program reads configuration data for the 5505 I/O module at address 5.

```
#include <ctools.h>

int main(void)
{
      UINT16      inputType[4];
      UINT16      inputFilter;
      BOOLEAN     status;
      BOOLEAN     done;

      // main loop
      while (TRUE)
```

```
      {
            // read output data from I/O table
            status = ioRead5505Outputs(5, inputType,
inputFilter);

            // print data when option switch 4 is selected
            if (optionSwitch(4))
            {
                  if (!done)
                  {
fprintf(com1, "status = %u,\
inputType 0 = %d, inputFilter = %d\r\n", status, inputType[0],
      inputFilter);

      done = TRUE;
                  }
            }
            else
            {
                  done = FALSE;
            }

            // release processor to other priority 1 tasks
            release_processor();
      }
}
```

ioRead5506Inputs

Read 5506 Inputs

Syntax

```
#include <ctools.h>
BOOLEAN ioRead5506Inputs(
      UINT16 moduleAddress,
      UCHAR &dinData,
      INT16 (&ainData)[8]
)
```

Description

This function reads buffered data from the digital and analog inputs of a 5506 I/O module. Buffered data are updated when an I/O request for the module is processed.

`moduleAddress` is the address of the 5506 module. Valid values are 0 to 15.

`dinData` is a reference to a UCHAR variable. Digital data for the 8 internal inputs are written to this variable. One bit in the variable represents each input point. The 8 internal inputs indicate if the corresponding analog input value is over range.

`ainData` is a reference to an array of eight `INT16` variables. Analog data are written to this array.

The function returns FALSE if the module address is invalid; otherwise TRUE is returned.

See Also

ioRead5506Outputs, **ioWrite5506Outputs**

Example

This program displays the values of the 8 internal digital inputs and the 5th analog input read from 5506 I/O at address 5.

```
#include <ctools.h>
#define     MY_EVENT    1

int main(void)
{
      UCHAR       dinData;
      INT16       ainData[8];
      IO_STATUS   io_status;
      BOOLEAN     status;
      BOOLEAN     done;

      // main loop
      while (TRUE)
      {
            // add module scan to queue
            if (!ioRequest(MT_5506Inputs, 5))
            {
                  status = FALSE;
            }
```

```
            // wait for scan to complete
            ioNotification(MY_EVENT);
            wait_event(MY_EVENT);

            // read input data from last scan
            status = ioRead5506Inputs(5, dinData, ainData);

            // check status of last scan
            if (!ioStatus(MT_5506Inputs, 5, &io_status))
            {
                  status = FALSE;
            }
            else if (!io_status.commStatus)
            {
                  status = FALSE;
            }

            // print data when option switch 4 is selected
            if (optionSwitch(4))
            {
                  if (!done)
                  {
                        fprintf(com1, "status = %u,\
                        Dins 0 to 7 = %X, Ain 4 = %d\r\n",
                        status, dinData, ainData[4]);
                        done = TRUE;
                  }
            }
            else
            {
                  done = FALSE;
            }

            // release processor to other priority 1 tasks
            release_processor();
      }
}
```

ioRead5506Outputs

Read 5506 Configuration

Syntax

```
#include <ctools.h>
BOOLEAN ioRead5506Outputs(
      UINT16 moduleAddress,
      UINT16 (&inputType)[8],
      UINT16 &inputFilter,
      UINT16 &scanFrequency
)
```

Description

This function reads configuration data from the I/O Table for a 5506 I/O module. Configuration data are written using the `ioWrite5506Outputs` function.

`moduleAddress` is the address of the 5506 module. Valid values are 0 to 15.

`inputType` is a reference to an array of eight UINT16 variables. Analog input measurement types are written to this array. Valid values are

- 0 = 0 to 5V
- 1 = 1 to 5 V
- 2 = 0 to 20 mA
- 3 = 4 to 20 mA.

`inputFilter` is a reference to a UINT16 variable. The input filter selection is written to this variable.

- 0 = 3 Hz
- 1 = 6 Hz
- 2 = 11 Hz
- 3 = 30 Hz

`scanFrequency` is a reference to a UINT16 variable. The scan frequency selection is written to this variable.

- 0 = 60 Hz
- 1 = 50 Hz

The function returns FALSE if the module address is invalid; otherwise TRUE is returned.

See Also

ioRead5506Inputs, **ioWrite5506Outputs**

Example

This program reads configuration data for the 5506 I/O module at address 5.

```
#include <ctools.h>

int main(void)
```

```
{
      UINT16      inputType[8];
      UINT16      inputFilter;
      UINT16      scanFrequency;
      BOOLEAN     status;
      BOOLEAN     done;

      // main loop
      while (TRUE)
      {
            // read output data from I/O table
            status = ioRead5506Outputs(5, inputType, inputFilter,
scanFrequency);

            // print data when option switch 4 is selected
            if (optionSwitch(4))
            {
                  if (!done)
                  {
fprintf(com1, "status = %u,\
inputType 0 = %d, inputFilter = %d,\
scanFrequency = %d \r\n",
                        status, inputType[0],
      inputFilter, scanFrequency);

      done = TRUE;
                  }
            }
            else
            {
                  done = FALSE;
            }

            // release processor to other priority 1 tasks
            release_processor();
      }
}
```

ioRead5606Inputs

Read 5606 Inputs

Syntax

```
#include <ctools.h>
BOOLEAN ioRead5606Inputs(
      UINT16 moduleAddress,
      UCHAR (&dinData)[5],
      INT16 (&ainData)[8]
)
```

Description

This function reads buffered data from the digital and analog inputs of a 5606 I/O module. Buffered data are updated when an I/O request for the module is processed.

`moduleAddress` is the address of the 5606 module. Valid values are 0 to 7.

`dinData` is a reference to an array of five UCHAR variables. Digital data for the 32 external and 8 internal inputs are written to this array. One bit in the array represents each input point. The 8 internal inputs indicate if the corresponding analog input value is over range.

`ainData` is a reference to an array of eight `INT16` variables. Analog data are written to this array.

The function returns FALSE if the module address is invalid; otherwise TRUE is returned.

See Also

ioRead5606Inputs, **ioRead5606Outputs**

Example

This program displays the values of the first 8 digital inputs and the 5th analog input read from 5606 I/O at address 5.

```
#include <ctools.h>
#define     MY_EVENT    1

int main(void)
{
      UCHAR       dinData[5];
      INT16       ainData[8];
      IO_STATUS   io_status;
      BOOLEAN     status;
      BOOLEAN     done;

      // main loop
      while (TRUE)
      {
            // add module scan to queue
            if (!ioRequest(MT_5606Inputs, 5))
            {
                  status = FALSE;
```

```
            }

            // wait for scan to complete
            ioNotification(MY_EVENT);
            wait_event(MY_EVENT);

            // read input data from last scan
            status = ioRead5606Inputs(5, dinData, ainData);

            // check status of last scan
            if (!ioStatus(MT_5606Inputs, 5, &io_status))
            {
                  status = FALSE;
            }
            else if (!io_status.commStatus)
            {
                  status = FALSE;
            }

            // print data when option switch 4 is selected
            if (optionSwitch(4))
            {
                  if (!done)
                  {
                        fprintf(com1, "status = %u,\
                        Dins 0 to 7 = %X, Ain 4 = %d\r\n",
                        status, dinData[0], ainData[4]);
                        done = TRUE;
                  }
            }
            else
            {
                  done = FALSE;
            }

            // release processor to other priority 1 tasks
            release_processor();
      }
}
```

ioRead5606Outputs

Read 5606 Outputs

Syntax

```
#include <ctools.h>
BOOLEAN ioRead5606Outputs(
      UINT16 moduleAddress,
      UCHAR (&doutData)[2],
      INT16 (&aoutData)[2],
      UINT16 (&inputType)[8],
      UINT16 &inputFilter,
      UINT16 &scanFrequency,
      UINT16 &outputType
)
```

Description

This function reads buffered data from the digital and analog outputs of a 5606 I/O module. Buffered data are written using the `ioWrite5606Outputs` function.

`moduleAddress` is the address of the 5606 module. Valid values are 0 to 7.

`doutData` is a reference to an array of two UCHAR variables. Digital data for the 16 outputs are written to this array. One bit in the array represents each output point.

`aoutData` is a reference to an array of two `INT16` variables. Analog data for the two analog outputs are written to this array.

`inputType` is a reference to an array of eight UINT16 variables. Analog input measurement types are written to this array. Valid values are

- 0 = 0 to 5V
- 1 = 0 to 10 V
- 2 = 0 to 20 mA
- 3 = 4 to 20 mA.

`inputFilter` is a reference to a UINT16 variable. The input filter selection is written to this variable.

- 0 = 3 Hz
- 1 = 6 Hz
- 2 = 11 Hz
- 3 = 30 Hz

`scanFrequency` is a reference to a UINT16 variable. The scan frequency selection is written to this variable.

- 0 = 60 Hz
- 1 = 50 Hz

`outputType` is a reference to a UINT16 variable. The analog output type is written to this variable.

- 0 = 0 to 20 mA
- 1 = 4 to 20 mA.

The function returns FALSE if the module address is invalid; otherwise TRUE is returned.

See Also

XXX ioRead5606Inputs, ioWrite5606Outputs

Example

This program reads output data from the I/O table for the 5606 digital outputs and analog outputs at address 5.

```
#include <ctools.h>

int main(void)
{
      UCHAR       doutData[2];
      INT16       aoutData[2];
      UINT16      inputType[8];
      UINT16      inputFilter;
      UINT16      scanFrequency;
      UINT16      outputType;
      BOOLEAN     status;
      BOOLEAN     done;

      // main loop
      while (TRUE)
      {
            // read output data from I/O table
            status = ioRead5606Outputs(5, doutData, aoutData,
inputType, inputFilter, scanFrequency, outputType);

            // print data when option switch 4 is selected
            if (optionSwitch(4))
            {
                  if (!done)
                  {
fprintf(com1, "status = %u,\
Douts 0 to 7 = %X, Aout 0 = %d\r\n",
                        status, doutData[0], aoutData[0]);

      done = TRUE;
                  }
            }
            else
            {
                  done = FALSE;
            }

            // release processor to other priority 1 tasks
            release_processor();
      }
}
```

ioReadAin4

Read Data From 4-point Analog Input Module

Syntax

```
#include <ctools.h>
BOOLEAN ioReadAin4(UINT16 moduleAddress, INT16 (&data)[4])
```

Description

This function reads buffered data from the 4 point analog input module at the specified module address. Buffered data are updated when an I/O request for the module is processed.

The function has two parameters: the module address, and a reference to an array of four INT16 variables. If the `moduleAddress` is valid, analog input data are copied to the array. The valid range for `moduleAddress` is 0 to 15.

The function returns FALSE if the module address is invalid; otherwise TRUE is returned.

ioReadAin8

Read Data From 8-point Analog Input Module

Syntax

```
#include <ctools.h>
BOOLEAN ioReadAin8(UINT16 moduleAddress, INT16 (&data)[8])
```

Description

This function reads buffered data from the 8 point analog input module at the specified `moduleAddress`. Buffered data are updated when an I/O request for the module is processed.

The function has two parameters: the module address, and a reference to an array of eight INT16 variables. If the `moduleAddress` is valid, analog input data are copied to the array. The valid range for `moduleAddress` is 0 to 15.

The function returns FALSE if the module address is invalid; otherwise TRUE is returned.

ioReadAout2

Read Data From 2-point Analog Output Module

Syntax

```
#include <ctools.h>
BOOLEAN ioReadAout2(UINT16 moduleAddress, INT16 (&data)[2])
```

Description

This function reads buffered data used for the 2-point analog output module at the specified module address. Buffered data are written using the `ioWriteAout2` function.

The function has two parameters: the module address, and a reference to an array of two INT16 variables. If the `moduleAddress` is valid, data are copied to the array. The valid range for `moduleAddress` is 0 to 15.

The function returns FALSE if the module address is invalid; otherwise TRUE is returned.

ioReadAout4

Read Data From 4-point Analog Output Module

Syntax

```
#include <ctools.h>
BOOLEAN ioReadAout4(UINT16 moduleAddress, INT16 (&data)[4])
```

Description

This function reads buffered data used for the 4-point analog output module at the specified module address. Buffered data are written using the `ioWriteAout4` function.

The function has two parameters: the module address, and a reference to an array of four INT16 variables. If the `moduleAddress` is valid, data are copied to the array. The valid range for `moduleAddress` is 0 to 15.

The function returns FALSE if the module address is invalid; otherwise TRUE is returned.

ioReadAout5303

Read Data From 2-point 5303 Analog Output Module

Syntax

```
#include <ctools.h>
BOOLEAN ioReadAout5303(INT16 (&data)[2])
```

Description

This function reads buffered data used for the 2-point 5303 analog output module. Buffered data are written using the `ioWriteAout5303` function.

The function has one parameter: a reference to an array of two INT16 variables. The buffered data are copied to the array.

The function always returns TRUE.

ioReadCounter4

Read Data From 4-point Counter Input Module

Syntax

```
#include <ctools.h>
BOOLEAN ioReadCounter4(UINT16 moduleAddress, UINT32 (&data)[4])
```

Description

This function reads buffered data from the 4 point counter input module at the specified module address. Buffered data are updated when an I/O request for the module is processed.

The function has two parameters: the module address, and a reference to an array of four UINT32 variables. If the `moduleAddress` is valid, data are copied to the array. The valid range for `moduleAddress` is 0 to 15.

The maximum count is 4,294,967,295. Counters roll back to 0 when the maximum count is exceeded.

The function returns FALSE if the module address is invalid; otherwise TRUE is returned.

ioReadCounterSP2

Read Data From the SCADAPack2 Counter Inputs

Syntax

```
#include <ctools.h>
BOOLEAN ioReadCounterSP2 (UINT32 (&data)[3])
```

Description

This function reads buffered data from the SCADAPack2 counter inputs. Buffered data are updated when an I/O request for the module is processed.

The function has one parameter: a reference to an array of three UINT32 variables. The buffered data are copied to the array.

The maximum count is 4,294,967,295. Counters roll back to 0 when the maximum count is exceeded.

The function always returns TRUE.

ioReadDin16

Read Data From 16-point Digital Input Module

Syntax

```
#include <ctools.h>
BOOLEAN ioReadDin16(UINT16 moduleAddress, UINT16 & data)
```

Description

This function reads buffered data from the 16 point digital input module at the specified module address. Buffered data are updated when an I/O request for the module is processed.

The function has two parameters: the module address, and a reference to an INT16 variable. If the `moduleAddress` is valid, digital input data are copied to the variable. The valid range for `moduleAddress` is 0 to 15.

The function returns FALSE if the module address is invalid; otherwise TRUE is returned.

ioReadDin32

Read 32 Digital Inputs

Syntax

```
#include <ctools.h>
BOOLEAN ioReadDin32(UINT16 moduleAddress, UINT32 & data)
```

Description

This function reads buffered data from the 32 point digital input module at the specified module address. Buffered data are updated when an I/O request for the module is processed.

`moduleAddress` is the address of the digital output module. The valid range is 0 to 15.

`data` is a reference to a variable to receive the input data.

The function returns TRUE if `data` was written. The function returns FALSE if the module address is invalid.

See Also

ioReadDin8, ioReadDin16

Example

This program displays the values of the 32 digital inputs read from a 32 point Digital Input Module at module address 0.

```
#include <ctools.h>

#define IO_NOTIFICATION 0

int main(void)
{
      UINT16 point;
      UINT32 dinData;

      /* request read from digital input module */
      ioRequest(MT_Din32, 0);

      /* wait for the read to complete */
      ioNotification(IO_NOTIFICATION);
      wait_event(IO_NOTIFICATION);

      /* get the data read */
      ioReadDin32(0, dinData);

      /* Print module data */
      fprintf(com1, "Point          Value");
      for (point = 0; point < 32; point++)
      {
            fprintf(com1, "\n\r%d   ", point);
            putchar(dinData & 0x0001 ? '1' :'0');
            dinData >>= 1;
```

```
            }
        }
```

ioReadDin8

Read Data From 8-point Digital Input Module

Syntax

```
#include <ctools.h>
BOOLEAN ioReadDin8(UINT16 moduleAddress, UCHAR & data)
```

Description

This function reads buffered data from the 8 point digital input module at the specified module address. Buffered data are updated when an I/O request for the module is processed.

The function has two parameters: the module address, and a reference to an UCHAR variable. If the `moduleAddress` is valid, digital input data are copied to the variable. The valid range for `moduleAddress` is 0 to 15.

The function returns FALSE if the module address is invalid; otherwise TRUE is returned.

ioReadDout16

Read Data From 16-point Digital Output Module

Syntax

```
#include <ctools.h>
BOOLEAN ioReadDout16(UINT16 moduleAddress, UINT16 & data)
```

Description

This function reads buffered data used for the 16-point digital output module at the specified module address. Buffered data are written using the `ioWriteDout16` function.

The function has two parameters: the module address, and a pointer to an UINT16 variable. If the `moduleAddress` is valid, digital input data are copied to the variable. The valid range for `moduleAddress` is 0 to 15.

The function returns FALSE if the module address is invalid; otherwise TRUE is returned.

ioReadDout32

Read from 32 Digital Outputs

Syntax

```
#include <ctools.h>
BOOLEAN ioReadDout32(
      UINT16 moduleAddress,
      UINT32 & data)
```

Description

The `ioReadDout32` function reads buffered data for a 32-bit digital output module. Buffered data are written using the `ioWriteDout32` function.

The function has two parameters.

`moduleAddress` is the address of the module. The valid range is 0 to 15.

`data` is reference to a UINT32 variable. If the module address is valid, data are copied to this variable.

The function returns FALSE if the `moduleAddress` is invalid; otherwise TRUE is returned.

See Also

ioReadDout8, ioReadDout16

ioReadDout8

Read Data From 8-point Digital Output Module

Syntax

```
#include <ctools.h>
BOOLEAN ioReadDout8(UINT16 moduleAddress, UCHAR & data)
```

Description

This function reads buffered data used for the 8-point digital output module at the specified module address. Buffered data are written using the `ioWriteDout8` function.

The function has two parameters: the module address, and a reference to an UCHAR variable. If the `moduleAddress` is valid, digital input data are copied to the variable. The valid range for `moduleAddress` is 0 to 15.

The function returns FALSE if the module address is invalid; otherwise TRUE is returned.

ioReadSP2Inputs

Read SCADAPack2 Inputs

Syntax

```
#include <ctools.h>
BOOLEAN ioReadSP2Inputs(
      UCHAR (&dinData)[2],
      INT16 (&ainData)[8]
)
```

Description

This function reads buffered data from the digital and analog inputs of the SCADAPack2 I/O. Buffered data are updated when an I/O request for the module is processed.

`dinData` is a reference to an array of two UCHAR variables. Digital data for the 12 inputs are written to this array. One bit in the array represents each input point.

`ainData` is a reference to an array of eight `INT16` variables. Analog data are written to this array.

The function returns FALSE if the module address is invalid; otherwise TRUE is returned.

See Also

`ioReadSP2Outputs, ioWriteSP2Outputs`

Example

This program displays the values of the first 8 digital inputs and the 5th analog input read from the SCADAPack2 I/O.

```
#include <ctools.h>
#define     MY_EVENT    1

int main(void)
{
      UCHAR       dinData[2];
      INT16       ainData[8];
      IO_STATUS   io_status;
      BOOLEAN     status;
      BOOLEAN     done;
      BOOLEAN     printNow;

      // main loop
      while (TRUE)
      {
            // add module scan to queue
            if (!ioRequest(MT_SP2Inputs, 0))
            {
                  status = FALSE;
            }

            // wait for scan to complete
            ioNotification(MY_EVENT);
```

```
            wait_event(MY_EVENT);

            // read input data from last scan
            status = ioReadSP2Inputs(dinData, ainData);

            // check status of last scan
            if (!ioStatus(MT_SP2Inputs, 0, &io_status))
            {
                  status = FALSE;
            }
            else if (!io_status.commStatus)
            {
                  status = FALSE;
            }

            // print data when coil register 100 is selected
            request_resource(IO_SYSTEM);
            printNow = dbase(MODBUS, 100);
            release_resource(IO_SYSTEM);
            if (printNow)
            {
                  if (!done)
                  {
                        fprintf(com1, "status = %u,\
                        Dins 0 to 7 = %X, Ain 4 = %d\r\n",
                        status, dinData[0], ainData[4]);
                        done = TRUE;
                  }
            }
            else
            {
                  done = FALSE;
            }

            // release processor to other priority 1 tasks
            release_processor();
      }
}
```

ioReadSP2Outputs

Read SCADAPack2 Outputs

Syntax

```
#include <ctools.h>
BOOLEAN ioReadSP2Outputs(
      UCHAR (&doutData)[2],
      INT16 (&aoutData)[2]
)
```

Description

This function reads buffered data from the digital and analog outputs of a SCADAPack2 I/O module. Buffered data are written using the `ioWriteSP2Outputs` function.

`doutData` is a reference to an array of two UCHAR variables. Digital data for the 10 outputs are written to this array. One bit in the array represents each output point.

`aoutData` is a reference to an array of two `INT16` variables. Analog data for the two analog outputs are written to this array.

The function always returns TRUE.

See Also

`ioReadSP2Inputs, ioWriteSP2Outputs`

Example

This program reads output data from the I/O table for the SCADAPack2 digital outputs and analog outputs.

```
#include <ctools.h>

int main(void)
{
      UCHAR       doutData[2];
      INT16       aoutData[2];
      BOOLEAN     status;
      BOOLEAN     done;
      BOOLEAN     printNow;

      // main loop
      while (TRUE)
      {
            // read output data from I/O table
            status = ioReadSP2Outputs(doutData, aoutData);

            // print data when coil register 100 is selected
            request_resource(IO_SYSTEM);
            printNow = dbase(MODBUS, 100);
            release_resource(IO_SYSTEM);
            if (printNow)
            {
                  if (!done)
                  {
```

```
                        fprintf(com1, "status = %u,\
                        Douts 0 to 7 = %X, Aout 0 = %d\r\n",
                        status, doutData[0], aoutData[0]);

                        done = TRUE;
                  }
            }
            else
            {
                  done = FALSE;
            }

            // release processor to other priority 1 tasks
            release_processor();
      }
}
```

ioRequest

Add I/O Module Scan Request to Request Queue

Syntax

```
#include <ctools.h>
BOOLEAN ioRequest(IO_TYPE moduleType, UINT16 moduleAddress)
```

Description

This function adds to the I/O Controller request queue an I/O module scan request for the specified I/O module.

The function has two arguments: the module type, and the module address. Refer to the table below for valid I/O module types and address ranges.

The function returns TRUE if the request was added. The function returns FALSE if there is no room in the request queue or if an argument is invalid.

I/O Module Type	Address Range
MT_Ain4	0 to 15
MT_Ain8	0 to 15
MT_Aout2	0 to 15
MT_Aout4	0 to 15
MT_Din8	0 to 15
MT_Din16	0 to 15
MT_Dout8	0 to 15
MT_Dout16	0 to 15
MT_Counter4	0 to 15
MT_5601Inputs	not applicable
MT_5601Outputs	not applicable
MT_5904Inputs	0 to 3
MT_5904Outputs	0 to 3
MT_CounterSP2	not applicable
MT_SP2Inputs	not applicable
MT_SP2Outputs	not applicable
MT_Dout32	0 to 15
MT_Din32	0 to 15
MT_5604Inputs	not applicable
MT_5604Outputs	not applicable
MT_Aout4_Checksum	0 to 15

ioSetConfiguration

Set I/O Controller Configuration

Syntax

```
#include <ctools.h>
BOOLEAN ioSetConfiguration(const IO_CONFIG & settings)
```

Description

This function sets the I/O controller configuration and adds a request to write the settings to the I/O controller.

The function has one argument: a reference to an IO_CONFIG structure.

The function returns TRUE if the request was added. The function returns FALSE if there is no room in the request queue or if there is an error in the settings.

ioStatus

Read Status of Last Scan of Specified I/O Module

Syntax

```
#include <ctools.h>
BOOLEAN ioStatus(IO_TYPE moduleType, UINT16 moduleAddress,
IO_STATUS * status)
```

Description

This function reads the status of the last scan of the specified I/O module.

The function has three arguments: the module type, the module address, and a pointer to an IO_STATUS structure. Refer to the table below for valid I/O module types and address ranges.

The function returns TRUE if status information was copied to the structure pointed to by *status*. The function returns FALSE if an argument is invalid.

I/O Module Type	Address Range
MT_Ain4	0 to 15
MT_Ain8	0 to 15
MT_Aout2	0 to 15
MT_Aout4	0 to 15
MT_Din8	0 to 15
MT_Din16	0 to 15
MT_Dout8	0 to 15
MT_Dout16	0 to 15
MT_Counter4	0 to 15
MT_5601Inputs	not applicable
MT_5601Outputs	not applicable
MT_5904Inputs	0 to 3
MT_5904Outputs	0 to 3
MT_CounterSP2	not applicable
MT_SP2Inputs	not applicable
MT_SP2Outputs	not applicable
MT_Dout32	0 to 15
MT_Din32	0 to 15
MT_5604Inputs	not applicable
MT_5604Outputs	not applicable
MT_Aout4_Checksum	0 to 15

ioSystemReset

Add Reset Request to I/O Controller Request Queue

Syntax

```
#include <ctools.h>
BOOLEAN ioSystemReset(void)
```

Description

This function adds a reset request to the I/O Controller request queue. When the request is sent to the I/O Controller, all I/O modules are reset.

The function has no arguments.

The function returns TRUE if the request was added. The function returns FALSE if there is no room in the request queue.

ioVersion

Get the I/O Controller Firmware Version

Syntax

```
#include <ctools.h>
BOOLEAN ioVersion(UINT16 & pVersion)
```

Description

This function returns the I/O controller firmware version. The version is read from the I/O controller at initialization.

The function has one argument: a reference to an UINT16 value to which the firmware version is copied if it is available.

The function returns TRUE if the firmware version is available. It returns FALSE if the firmware version has not been read from the I/O controller.

ioWrite5505Outputs

Write 5505 Configuration

Syntax

```
#include <ctools.h>
BOOLEAN ioWrite5505Outputs(
      UINT16 moduleAddress,
      UINT16 (&inputType)[4],
      UINT16 inputFilter
)
```

Description

This function writes configuration data to the I/O Table for a 5505 I/O module. Data are written to the module when an I/O request for the module is processed.

`moduleAddress` is the address of the 5505 module. Valid values are 0 to 15.

`inputType` is a reference to an array of four UINT16 variables selecting the input range for the corresponding analog input. Valid values are

- 0 = RTD in deg Celsius
- 1 = RTD in deg Fahrenheit
- 2 = RTD in deg Kelvin
- 3 = resistance measurement in ohms.

`inputFilter` selects input filter selection is written to this variable.

- 0 = 0.5 s
- 1 = 1 s
- 2 = 2 s
- 3 = 4 s

The function returns FALSE if the module address is invalid; otherwise TRUE is returned.

See Also

ioRead5505Inputs, **ioRead5505Outputs**

Example

This program writes configuration data to the 5505 module at address 5.

```
#include <ctools.h>
#define     MY_EVENT    1

int main(void)
{
      UINT16      inputType[4];
      UINT16      inputFilter;
      IO_STATUS   io_status;
      BOOLEAN     status;

      // main loop
      while (TRUE)
```

```
        {
                /* set analog input types to RTD in deg F */
                inputType[0] = 1;
                inputType[1] = 1;
                inputType[2] = 1;
                inputType[3] = 1;

                /* set filter */
                inputFilter = 3;        // minimum filter

                status = ioWrite5505Outputs(5, inputType,
inputFilter);

                // add module scan to queue
                if (!ioRequest(MT_5505Outputs, 5))
                {
                        status = FALSE;
                }

                // wait for scan to complete
                ioNotification(MY_EVENT);
                wait_event(MY_EVENT);

                // check status of last scan
                if (!ioStatus(MT_5505Outputs, 5, &io_status))
                {
                        status = FALSE;
                }
                else if (!io_status.commStatus)
                {
                        status = FALSE;
                }

                // release processor to other priority 1 tasks
                release_processor();
        }
}
```

ioWrite5506Outputs

Write 5506 Configuration

Syntax

```
#include <ctools.h>
BOOLEAN ioWrite5506Outputs(
      UINT16 moduleAddress,
      UINT16 (&inputType)[8],
      UINT16 inputFilter,
      UINT16 scanFrequency
)
```

Description

This function writes configuration data to the I/O Table for a 5506 I/O module. Data are written to the module when an I/O request for the module is processed.

`moduleAddress` is the address of the 5506 module. Valid values are 0 to 15.

`inputType` is a reference to an array of eight UINT16 variables selecting the input range for the corresponding analog input. Valid values are

- 0 = 0 to 5V
- 1 = 1 to 5 V
- 2 = 0 to 20 mA
- 3 = 4 to 20 mA.

`inputFilter` selects input filter selection is written to this variable.

- 0 = 3 Hz
- 1 = 6 Hz
- 2 = 11 Hz
- 3 = 30 Hz

`scanFrequency` selects the scan frequency setting. Valid values are.

- 0 = 60 Hz
- 1 = 50 Hz

The function returns FALSE if the module address is invalid; otherwise TRUE is returned.

See Also

ioRead5506Inputs, **ioRead5506Outputs**

Example

This program writes configuration data to the 5506 module at address 5.

```
#include <ctools.h>
#define     MY_EVENT    1

int main(void)
{
```

```
        UINT16      inputType[8];
        UINT16      inputFilter;
        UINT16      scanFrequency;
        IO_STATUS   io_status;
        BOOLEAN     status;

        // main loop
        while (TRUE)
        {
              /* set analog input types to 4-20 mA */
              inputType[0] = 3;
              inputType[1] = 3;
              inputType[2] = 3;
              inputType[3] = 3;
              inputType[4] = 3;
              inputType[5] = 3;
              inputType[6] = 3;
              inputType[7] = 3;

              /* set filter and frequency */
              inputFilter = 3;        // minimum filter
              scanFrequency = 0;      // 60 Hz

              status = ioWrite5506Outputs(5, inputType,
inputFilter, scanFrequency);

              // add module scan to queue
              if (!ioRequest(MT_5506Outputs, 5))
              {
                    status = FALSE;
              }

              // wait for scan to complete
              ioNotification(MY_EVENT);
              wait_event(MY_EVENT);

              // check status of last scan
              if (!ioStatus(MT_5506Outputs, 5, &io_status))
              {
                    status = FALSE;
              }
              else if (!io_status.commStatus)
              {
                    status = FALSE;
              }

              // release processor to other priority 1 tasks
              release_processor();
        }
}
```

ioWrite5606Outputs

Write 5606 Outputs

Syntax

```
#include <ctools.h>
BOOLEAN ioWrite5606Outputs(
      UINT16 moduleAddress,
      UCHAR (&doutData)[2],
      INT16 (&aoutData)[2],
      UINT16 (&inputType)[8],
      UINT16 inputFilter,
      UINT16 scanFrequency,
      UINT16 outputType
)
```

Description

This function writes data to the I/O table for the 16 digital outputs and 2 analog outputs of a 5606 I/O module. Data are written to the module when an I/O request for the module is processed.

`moduleAddress` is the address of the 5606 module. Valid values are 0 to 7.

`doutData` is a reference to an array of two UCHAR variables. Digital data for the 16 outputs are read from this array. One bit in the array represents each output point.

`aoutData` is a reference to an array of two `INT16` variables. Analog data for the two analog outputs are read from this array.

`inputType` is a reference to an array of eight UINT16 variables selecting the input range for the corresponding analog input. Valid values are

- 0 = 0 to 5V
- 1 = 0 to 10 V
- 2 = 0 to 20 mA
- 3 = 4 to 20 mA.

`inputFilter` selects input filter selection is written to this variable.

- 0 = 3 Hz
- 1 = 6 Hz
- 2 = 11 Hz
- 3 = 30 Hz

`scanFrequency` selects the scan frequency setting. Valid values are.

- 0 = 60 Hz
- 1 = 50 Hz

`outputType` selects the analog output type setting. Valid values are.

- 0 = 0 to 20 mA
- 1 = 4 to 20 mA.

The function returns FALSE if the module address is invalid; otherwise TRUE is returned.

See Also

ioRead5606Inputs, **ioRead5606Outputs**

Example

This program turns on all 16 digital outputs and sets the analog outputs to full scale on the 5606 module at address 5.

```
#include <ctools.h>
#define     MY_EVENT    1

int main(void)
{
      UCHAR       doutData[2];
      INT16       aoutData[2];
      UINT16      inputType[8];
      UINT16      inputFilter;
      UINT16      scanFrequency;
      UINT16      outputType;
      IO_STATUS   io_status;
      BOOLEAN     status;

      // main loop
      while (TRUE)
      {
            // write data to output tables for next scan
            doutData[0] = 0xFF;
            doutData[1] = 0xFF;
            aoutData[0] = 32767;
            aoutData[1] = 32767;

            /* set analog input types to 4-20 mA */
            inputType[0] = 3;
            inputType[1] = 3;
            inputType[2] = 3;
            inputType[3] = 3;
            inputType[4] = 3;
            inputType[5] = 3;
            inputType[6] = 3;
            inputType[7] = 3;

            /* set filter and frequency */
            inputFilter = 3;        // minimum filter
            scanFrequency = 0;      // 60 Hz

      /* set analog output type to 4-20 mA */
      outputType = 1;

            status = ioWrite5606Outputs(5, doutData, aoutData,
inputType, inputFilter, scanFrequency, outputType);

            // add module scan to queue
            if (!ioRequest(MT_5606Outputs, 5))
            {
                  status = FALSE;
            }
```

```
            // wait for scan to complete
            ioNotification(MY_EVENT);
            wait_event(MY_EVENT);

            // check status of last scan
            if (!ioStatus(MT_5606Outputs, 5, &io_status))
            {
                  status = FALSE;
            }
            else if (!io_status.commStatus)
            {
                  status = FALSE;
            }

            // release processor to other priority 1 tasks
            release_processor();
      }
}
```

ioWriteAout2

Write Data to 2-Point Analog Output Module

Syntax

```
#include <ctools.h>
BOOLEAN ioWriteAout2(UINT16 moduleAddress, INT16 (&data)[2])
```

Description

This function writes data to the I/O tables for the 2-point analog output module at the specified module address. Data are written to the module when an I/O request for the module is processed.

The function has two parameters: the module address, and a reference to an array of two INT16 variables. Data are read from the array and written to the I/O table. The valid range for `moduleAddress` is 0 to 15.

The function returns TRUE if the data was written. The function returns FALSE if the module address is invalid.

ioWriteAout4

Write Data to 4-Point Analog Output Module

Syntax

```
#include <ctools.h>
BOOLEAN ioWriteAout4(UINT16 moduleAddress, INT16 (&data)[4])
```

Description

This function writes data to the I/O tables for the 4-point analog output module at the specified module address. Data are written to the module when an I/O request for the module is processed.

The function has two parameters: the module address, and a reference to an array of four INT16 variables. Data are read from the array and written to the I/O table. The valid range for `moduleAddress` is 0 to 15.

The function returns TRUE if the data was written. The function returns FALSE if the module address is invalid.

Notes

This function writes to the output table only. Use the `ioRequest` function to write the data to the module.

- Call `ioRequest` with the module type `MT_Aout4` for analog output modules without checksum support. All modules can use this module type.
- Call `ioRequest` with the module type `MT_Aout4_Checksum` for analog output modules with checksum support. Some modules such as the 5304 can use this module type.

Example

This example sets all four outputs of any analog output module to half scale.

```
#include <ctools.h>

int main(void)
{
      INT16 dataArray[4];

      /* set all output values to one-half scale */
      dataArray[0] = 16384;
      dataArray[1] = 16384;
      dataArray[2] = 16384;
      dataArray[3] = 16384;

      /* Write data to analog output module at
        module address 0 */
      ioWriteAout4(0, dataArray);
      ioRequest(MT_Aout4, 0);
}
```

ioWriteAout5303

Write Data to 5303 Analog Output Module

Syntax

```
#include <ctools.h>
BOOLEAN ioWriteAout5303(INT16 (&data)[2])
```

Description

This function writes data to the I/O tables for the 2-point 5303 analog output module. Data are written to the module when an I/O request for the module is processed.

The function has one parameter: a reference to an array of two INT16 variables. Data are read from the array and written to the I/O table.

The function always returns TRUE.

ioWriteDout16

Write Data to 16-Point Digital Output Module

Syntax

```
#include <ctools.h>
BOOLEAN ioWriteDout16(UINT16 moduleAddress, UINT16 data)
```

Description

This function writes data to the I/O tables for the 16-point digital output module at the specified module address. Data are written to the module when an I/O request for the module is processed.

The function has two parameters: the module address, and the data to be written. Data are read from the 16-bit *data* value and written to the I/O table. The valid range for `moduleAddress` is 0 to 15.

The function returns TRUE if the data was written. The function returns FALSE if the module address is invalid.

ioWriteDout32

Write to 32 Digital Outputs

Syntax

```
#include <ctools.h>
BOOLEAN ioWriteDout32(
      UINT16 moduleAddress,
      UINT32 data)
```

Description

This function writes data to the I/O tables for the 32-point digital output module at the specified module address. Data are written to the module when an I/O request for the module is processed.

`moduleAddress` is the address of the digital output module. The valid range is 0 to 15.

`data` is the output data to be written. Data are written to the I/O table.

The function returns TRUE if the data was written. The function returns FALSE if the module address is invalid.

See Also

Example

This program turns ON all 32 digital outputs of a 32-point Digital Output Module at module address 0.

```
#include <ctools.h>

int main(void)
{
      /* Write data to digital output module */
      ioWriteDout32(0, 0xFFFFFFFF);
      ioRequest(MT_Dout32, 0);
}
```

ioWriteDout8

Write Data to 8-Point Digital Output Module

Syntax

```
#include <ctools.h>
BOOLEAN ioWriteDout8(UINT16 moduleAddress, UCHAR data)
```

Description

This function writes data to the I/O tables for the 8-point digital output module at the specified module address. Data are written to the module when an I/O request for the module is processed.

The function has two parameters: the module address, and the data to be written. Data are read from the 8-bit *data* value and written to the I/O table. The valid range for `moduleAddress` is 0 to 15.

The function returns TRUE if the data was written. The function returns FALSE if the module address is invalid.

ioWriteSP2Outputs

Write SCADAPack2 Outputs

Syntax

```
#include <ctools.h>
BOOLEAN ioWriteSP2Outputs(
      UCHAR (&doutData)[2],
      INT16 (&aoutData)[2]
)
```

Description

This function writes data to the I/O table for the 10 digital outputs and 2 analog outputs of the SCADAPack2 I/O. Data are written to the module when an I/O request for the module is processed.

`doutData` is a reference to an array of two UCHAR variables. Digital data for the 10 outputs are read from this array. One bit in the array represents each output point.

`aoutData` is a reference to an array of two `INT16` variables. Analog data for the two analog outputs are read from this array.

The function always returns TRUE.

See Also

`ioReadSP2Outputs, ioReadSP2Inputs`

Example

This program turns on all 10 digital outputs and sets the analog outputs to full scale on the SCADAPack2.

```
#include <ctools.h>
#define     MY_EVENT    1

int main(void)
{
      UCHAR       doutData[2];
      INT16       aoutData[2];
      IO_STATUS   io_status;
      BOOLEAN     status;

      // main loop
      while (TRUE)
      {
            // write data to output tables for next scan
            doutData[0] = 0xFF;
            doutData[1] = 0x03;
            aoutData[0] = 32767;
            aoutData[1] = 32767;

            status = ioWriteSP2Outputs(doutData, aoutData);

            // add module scan to queue
            if (!ioRequest(MT_SP2Outputs, 0))
```

```
                {
                        status = FALSE;
                }

                // wait for scan to complete
                ioNotification(MY_EVENT);
                wait_event(MY_EVENT);

                // check status of last scan
                if (!ioStatus(MT_SP2Outputs, 0, &io_status))
                {
                        status = FALSE;
                }
                else if (!io_status.commStatus)
                {
                        status = FALSE;
                }

                // release processor to other priority 1 tasks
                release_processor();
        }
}
```

ipFindFriendlyIPAddress

Checks if an address is in the Friendly IP List

Syntax

```
BOOLEAN ipFindFriendlyIPAddress(UINT32 ipAddress);
```

Description

This function checks if the IP address `ipAddress` is in the Friendly IP List.

The function returns TRUE if the supplied `ipAddress` is in the Friendly IP List. Otherwise FALSE is returned.

ipGetConnectionSummary

Get Summary of Active TCP/IP Connections

Syntax

```
#include <ctools.h>
void ipGetConnectionSummary( IP_CONNECTION_SUMMARY * pSummary );
```

Description

The **ipGetConnectionSummary** function returns a summary of the number of active IP connections. The IP connections include Modbus/TCP, Modbus RTU over UDP, Modbus ASCII over UDP, DNP over TCP, and DNP over UDP. The information is copied to the structure pointed to be *pSummary*. The structure IP_CONNECTION_SUMMARY is described in the *Structures and Types* section.

The information in the structure summarizes the number of connections as: master, slave or unused. Note that if a connection is allocated to master messaging but is currently disconnected, it will still be listed in the number of master connections.

Also, additional connections for store and forward translations will be included in the summary. For example, a master connection will be listed if a serial to Ethernet store and forward translation is currently active.

ipGetInterfaceType

Get Interface Type from IP Address

Syntax

```
#include <ctools.h>
BOOLEAN ipGetInterfaceType( IP_ADDRESS localIP, COM_INTERFACE * pIfType
);
```

Description

The function **ipGetInterfaceType** determines the interface that is configured to the specified local IP address, *localIP*. If no interface is configured to the specified IP address FALSE is returned; otherwise TRUE is returned and the interface type if copied to the value pointed to by *ifType*.

ipInitializeFriendlyIPSettings

Reset the friendly IP list

Syntax

```
void ipInitializeFriendlyIPSettings(void);
```

Description

This function deletes all Friendly IP List entries and disables the Friendly IP List.

The function has no parameters.

The function has no return value.

ipReadFriendlyListControl

Get the status of the friendly IP list

Syntax

```
UCHAR ipReadFriendlyListControl(void);
```

Description

This function returns the status of friendly IP list control.

The function has no parameters.

The function returns TRUE if friendly IP list is enabled and FALSE otherwise.

See Also

ipWriteFriendlyListControl

ipReadFriendlyIPListEntry

Read one entry in the friendly IP list

Syntax

```
BOOLEAN ipIpReadFriendlyIPListEntry(
UINT16 index,
IP_ADDRESS *pIpAddressStart
IP_ADDRESS *pIpAddressEnd
);
```

Description

This function reads an entry from the Friendly IP List.

`index` specifies the location in the list, and must be less than or equal to the Friendly IP List size.

`pIpAddressStart` and `pIpAddressStart` are pointers to IP addresses; they are written by this function.

The function returns TRUE if successful and FALSE if the index is invalid.

See Also

ipReadFriendlyIPListSize, ipWriteFriendlyIPListEntry, ipWriteFriendlyIPListSize

ipReadFriendlyIPListSize

Read the size of the friendly IP list

Syntax

```
UINT16 ipReadFriendlyIPListSize(void);
```

Description

This function reads the total number of active entries in the Friendly IP List.

The function has no parameters.

The function returns the total number of active entries in the list or zero if the list is empty.

See Also

ipReadFriendlyIPListEntry, ipWriteFriendlyIPListEntry, ipWriteFriendlyIPListSize

ipWriteFriendlyListControl

Enable or disable the friendly IP list

Syntax

```
BOOLEAN ipWriteFriendlyListControl(
      BOOLEAN state
      );
```

Description

This function enables or disables the friendly IP list. When the list is disabled the controller accepts messages from any IP address. When the list is enabled only messages from the IP addresses on the list are accepted.

`state` specifies if the friendly IP list is enabled or disabled. Valid values are TRUE (enabled) and FALSE (disabled). If the list is not valid then it can not be enabled.

The function returns TRUE if command was successful. It returns FALSE if it was attempted to enable an empty list or a list with invalid entries.

See Also

ipReadFriendlyListControl

ipWriteFriendlyIPListEntry

Write one entry in the friendly IP list

Syntax

```
BOOLEAN ipWriteFriendlyIPListEntry(
      UINT16 index,
      IP_ADDRESS ipAddressStart,
      IP_ADDRESS ipAddressEnd
      );
```

Description

This function writes an entry in the Friendly IP List.

`index` specifies the location in the list, and must be less than or equal to the Friendly IP List size.

`ipAddressStart` and `ipAddressEnd` specify a range of IP addresses (or a single IP address if they are the same) to be added to the list. Valid values are any IP address; the start IP address must be lower than or equal to the end IP address.

The function returns TRUE if successful and FALSE if the index or address is invalid.

Notes

`IpWriteFriendlyIPListSize` must be called before calling this function.

See Also

ipReadFriendlyIPListEntry

ipWriteFriendlyIPListSize

Write the size of the Friendly IP List

Syntax

```
BOOLEAN ipWriteFriendlyIPListSize(UINT16 size);
```

Description

This function sets the size of the Friendly IP List. This must be written before any entries are written to the list.

`size` specifies the number of active entries in the list. Valid values are 0 to 32.

The function returns TRUE if successful, FALSE otherwise.

See Also

ipReadFriendlyIPListSize

ledGetDefault

Read LED Power Control Parameters

Syntax

```
#include <ctools.h>
struct ledControl_tag ledGetDefault(void);
```

Description

The **ledGetDefault** routine returns the default LED power control parameters. The controller controls LED power to 5000 series I/O modules. To conserve power, the LEDs can be disabled.

The user can change the LED power setting with the LED POWER switch on the controller. The LED power returns to its default state after a user specified time period.

Example

See the example for the **ledSetDefault** function.

ledPower

Set LED Power State

Syntax

```
#include <ctools.h>
UINT16 ledPower(UINT16 state);
```

Description

The **ledPower** function sets the LED power state. The LED power will remain in the state until the default time-out period expires. *state* must be LED_ON or LED_OFF.

The function returns TRUE if state is valid and FALSE if it is not.

Notes

The LED POWER switch also controls the LED power. A user may override the setting made by this function.

The **ledSetDefault** function sets the default state of the LED power. This state overrides the value set by this function.

See Also

ledPowerSwitch ledPowerSwitch

ledPowerSwitch

Read State of the LED Power Switch

Syntax

```
#include <ctools.h>
UINT16 ledPowerSwitch(void);
```

Description

The ledPowerSwitch function returns the status of the led power switch. The function returns FALSE if the switch is released and TRUE if the switch is pressed.

Notes

This switch may be used by the program for user input. However, pressing the switch will have the side effect of changing the LED power state.

See Also

ledPower, ledSetDefault

ledSetDefault

Set Default Parameters for LED Power Control

Syntax

```
#include <ctools.h>
UINT16 ledSetDefault(struct ledControl_tag ledControl);
```

Description

The **ledSetDefault** routine sets default parameters for LED power control. The controller controls LED power to 5000 series I/O modules. To conserve power, the LEDs can be disabled.

The LED power setting can be changed by the user with the LED POWER switch on the controller. The LED power returns to its default state after a user specified time period.

The *ledControl* structure contains the default values. Refer to the **Structures and Types** section for a description of the fields in the *ledControl_tag* structure. Valid values for the *state* field are LED_ON and LED_OFF. Valid values for the *time* field are 1 to 65535 minutes.

The function returns TRUE if the parameters are valid and false if they are not. If either parameter is not valid, the default values are not changed.

The IO_SYSTEM resource must be requested before calling this function.

To save these settings with the controller settings in flash memory so that they are loaded on controller reset, call **flashSettingsSave** as shown below.

```
request_resource(FLASH_MEMORY);
flashSettingsSave(CS_RUN);
release_resource(FLASH_MEMORY);
```

Example

```
#include <ctools.h>

int main(void)
{
       struct ledControl_tag ledControl;

       request_resource(IO_SYSTEM);

       /* Turn LEDS off after 20 minutes */
       ledControl.time  = 20;
       ledControl.state = LED_OFF;
       ledSetDefault(ledControl);

       release_resource(IO_SYSTEM);

       /* ... the reset of the program */
}
```

listen

Syntax

```
#include <ctools.h>
int listen
(
      int socketDescriptor,
      int backLog
);
```

Function Description

To accept connections, a socket is first created with **socket** a backlog for incoming connections is specified with **listen** and then the connections are accepted with **accept**. The **listen** call applies only to sockets of type SOCK_STREAM. The *backLog* parameter defines the maximum length the queue of pending connections may grow to. If a connection request arrives with the queue full, and the underlying protocol supports retransmission, the connection request may be ignored so that retries may succeed. For AF_INET sockets, the TCP will retry the connection. If the backlog is not cleared by the time the TCP times out, **connect** will fail with ETIMEDOUT.

Parameters

socketDescriptor	The socket descriptor to listen on.
backlog	The maximum number of outstanding connections allowed on the socket.

Returns

0	Success
-1	An error occurred.

listen can fail for the following reason:

EADDRINUSE	The address is currently used by another socket.
EBADF	The socket descriptor is invalid.
EOPNOTSUPP	The socket is not of a type that supports the operation listen.

master_message

Send Protocol Command

Syntax

```
#include <ctools.h>
UINT16 master_message(UCHAR port, UINT16 function, UINT16 slave_station,
       UINT16 slave_address, UINT16 master_address, UINT16 length);
```

Description

The **master_message** function sends a command using a communication protocol. The communication protocol task waits for the response from the slave station. The current task continues execution.

- *port* specifies the serial port.
- *function* specifies the protocol function code. Refer to the communication protocol manual for supported function codes.
- *slave* specifies the network address of the slave station. This is also known as the slave station number.
- *address* specifies the location of data in the slave station. Depending on the protocol function code, data may be read or written at this location.
- *master_address* specifies the location of data in the master (this controller). Depending on the protocol function code, data may be read or written at this location.
- *length* specifies the number or registers.

The **master_message** function returns the command status from the protocol driver.

Value	Description
MM_SENT	message transmitted to slave
MM_BAD_FUNCTION	function is not recognized
MM_BAD_SLAVE	slave station number is not valid
MM_BAD_ADDRESS	slave or master database address not valid
MM_BAD_LENGTH	too many or too few registers specified
MM_EXCEPTION_FUNCTION	Master message status: Modbus slave returned a function exception.
MM_EXCEPTION_ADDRESS	Master message status: Modbus slave returned an address exception.
MM_EXCEPTION_VALUE	Master message status: Modbus slave returned a value exception.

The calling task monitors the status of the command sent using the **get_protocol_status** function. The `command` field of the `prot_status` structure is set to **MM_SENT** if a master message is sent. It will be set to **MM_RECEIVED** when the response to the message is received.

The command status will be set to **MM_RSP_TIMEOUT** if the response is not received within 10 seconds. Sending a retry master message before this timeout will abort the previous message. To use a timeout other than 10 seconds, use the **serialModbusMaster** function.

The **master_message** function may be used at the same time on the same serial port as a TelePACE MSTR element or ISaGRAF master function block.

Notes

Refer to the communication protocol manual for more information.

Users of TeleSAFE BASIC and the TeleSAFE 6000 C compiler should note that the address parameter now specifies the actual database address, when used with the Modbus protocol. This parameter specified the address offset on these older TeleSAFE products.

The IO_SYSTEM resource must be requested before calling this function.

See Also

modbusExceptionStatus modbusSlaveID

Example

See the example in the Example Programs chapter under the section **Master Message Example Using Modbus Protocol.**

modbusExceptionStatus

Set Response to Protocol Command

Syntax

```
#include <ctools.h>
void modbusExceptionStatus(UCHAR status);
```

Description

The **modbusExceptionStatus** function is used in conjunction with the Modbus compatible communication protocol. It sets the result returned in response to the Read Exception Status command. This command is provided for compatibility with some Modbus protocol drivers for host computers.

The value of *status* is determined by the requirements of the host computer.

Notes

The specified result will be sent each time that the protocol command is received, until a new result is specified.

The result is cleared when the controller is reset. The application program must initialize the status each time it is run.

See Also

master_message

modbusSlaveID

Set Response to Protocol Command

Syntax

```
#include <ctools.h>
void modbusSlaveID(UCHAR *string, UINT16 length);
```

Description

The **modbusSlaveID** function is used in conjunction with the Modbus compatible communication protocol. It sets the result returned in response to the Report Slave ID command. This command is provided for compatibility with some Modbus protocol drivers for host computers.

string points to a string of at least *length* characters. The contents of the string are determined by the requirements of the host computer. The string is not NULL terminated and may contain multiple NULL characters.

The *length* specifies how many characters are returned by the protocol command. *length* must be in the range 1 to **REPORT_SLAVE_ID_SIZE**. If *length* is too large only the first **REPORT_SLAVE_ID_SIZE** characters of the string will be sent in response to the command.

Notes

The specified result will be sent each time that the protocol command is received, until a new result is specified.

The function copies the data pointed to by *string*. *string* may be modified after the function is called.

The result is cleared when the controller is reset. The application program must initialize the salve ID string each time it is run.

modemAbort

Unconditionally Terminate Dial-up Connection

Syntax

```
#include <ctools.h>
void modemAbort(FILE *port);
```

Description

The **modemAbort** function unconditionally terminates a dial-up connection, connection in progress or modem initialization started by the C application. *port* specifies the serial port where the modem is installed.

The connection or initialization is terminated only if it was started from a C application. Connections made from a Ladder Logic application and answered calls are not terminated.

This function can be used in a task exit handler.

Notes

The serial port type must be set to RS232_MODEM.

Note that a pause of a few seconds is required between terminating a connection and initiating a new call. This pause allows the external modem time to hang up.

Use this function in a task exit handler to clean-up any open dial-up connections or modem initializations. If a task is ended by executing end_task from another task, modem connections or initializations must be aborted in the exit handler. Otherwise, the reservation ID for the port remains valid. No other task or Ladder Logic program may use modem functions on the port. Failing to call **modemAbort** or **modemAbortAll** in the task exit handler may result in the port being unavailable to any programs until the controller is reset.

The modem connection or initialization is automatically terminated when ISaGRAF stops the C application and when the controller is rebooted.

All reservation IDs returned by the **modemDial** and **modemInit** functions on this port are invalid after calling **modemAbort**.

See Also

modemAbortAll, modemDial,

Example

Refer to the examples in the **Functions Overview** section.

modemAbortAll

Unconditionally Terminate All Dial-up Connections

Syntax

```
#include <ctools.h>
void modemAbortAll(void);
```

Description

The **modemAbortAll** function unconditionally terminates all dial-up connections, connections in progress or modem initializations started by the C application.

The connections or initializations are terminated only if they were started from a C application. Connections made from a Ladder Logic application and answered calls are not terminated.

This function can be used in a task exit handler.

Notes

Note that a pause of a few seconds is required between terminating a connection and initiating a new call. This pause allows the external modem time to hang up.

Use this function in a task exit handler to clean-up any open dial-up connections or modem initializations. If executing end_task from another task ends a task, modem connections or initializations must be aborted in the exit handler. Otherwise, the reservation ID for the port remains valid. No other task or Ladder Logic program may use modem functions on the port. Failing to call **modemAbort** or **modemAbortAll** in the task exit handler may result in the port being unavailable to any programs until the controller is reset.

The modem connection or initialization is automatically terminated when ISaGRAF stops the C application and when the controller is rebooted.

This function will terminate all open dial-up connections or modem initializations started by the C application - even those started by other tasks. The exit handler can safely call this function instead of multiple calls to **modemAbort** if all the connections or initializations were started from the same task.

All reservation IDs returned by the **modemDial** and **modemInit** functions are invalid after calling **modemAbort** or **modemAbortAll**.

See Also

Example

This program installs an exit handler for the main task that terminates any dial-up connections made by the task. This handler is not strictly necessary if ISaGRAF ends the main task. However, it demonstrates how to use the modemAbortAll function and an exit handler for another task in a more complex program.

```
#include <ctools.h>

/* ------------------------------------------
   The shutdown function aborts any active
   modem connections when the task is ended.
   ------------------------------------------ */
```

```
void shutdown(void)
{
      modemAbortAll();
}

int main(void)
{
      TASKINFO taskStatus;

      /* set up exit handler for this task */
      getTaskInfo(0, &taskStatus);
      installExitHandler(taskStatus.taskID, shutdown);

      while(TRUE)
      {
         /* rest of main task here */

         /* Allow other tasks to execute */
         release_processor();
      }
}
```

modemDial

Connect to a Remote Dial-up Controller

Syntax

```
#include <ctools.h>
enum DialError modemDial(struct ModemSetup *configuration, reserve_id
   *id);
```

Description

The **modemDial** function connects a controller to a remote controller using an external dial-up modem. One **modemDial** function may be active on each serial port. The **modemDial** function handles all port sharing and multiple dialing attempts.

The *ModemSetup* structure specified by *configuration* defines the serial port, dialing parameters, modem initialization string and the phone number to dial. Refer to the **Structures and Types** section for a description of the fields in the *ModemSetup* structure.

id points to a reservation identifier for the serial port. The identifier ensures that no other modem control function can access the serial port. This parameter must be supplied to the **modemDialEnd** and **modemDialStatus** functions.

The function returns an error code. DE_NoError indicates that the connect operation has begun. Any other code indicates an error. Refer to the *dialup.h* section for a complete description of error codes.

Notes

The serial port type must be set to RS232_MODEM.

The **modemDialStatus** function returns the status of the connection attempt initiated by modemDial.

The **modemDialEnd** function terminates the connection to the remote controller. Note that a pause of a few seconds is required between terminating a connection and initiating a new call. This pause allows the external modem time to hang up.

If a communication protocol is active on the serial port when a connection is initiated, the protocol will be disabled until the connection is made, then re-enabled. This allows the controller to communicate with the external modem on the port. The protocol settings will also be restored when a connection is terminated with the **modemDialEnd** function.

If a **modemInit** function or an incoming call is active on the port, the **modemDial** function cannot access the port and will return an error code of DE_NotInControl. If communication stops for more than five minutes, then outgoing call requests are allowed to end the incoming call. This prevents problems with the modem or the calling application from permanently disabling outgoing calls.

The reservation identifier is valid until the call is terminated and another modem function or an incoming call takes control of the port.

To optimize performance, minimize the length of messages on com3. Examples of recommended uses for com3 are for local operator display terminals, and for programming and diagnostics using the ISaGRAF program.

Do not call this function in a task exit handler.

Example

Refer to the examples in the Connecting with a Remote Controller Example section.

modemDialEnd

Terminate Dial-up Connection

Syntax

```
#include <ctools.h>
void modemDialEnd(FILE *port, reserve_id id, enum DialError *error);
```

Description

The **modemDialEnd** function terminates a dial-up connection or connection in progress. *port* specifies the serial port the where the modem is installed. *id* is the port reservation identifier returned by the modemDial function.

The function sets the variable pointed to by *error*. If no error occurred DE_NoError is returned. Any other value indicates an error. Refer to the **Structures and Types** section for a complete description of error codes.

Notes

The serial port type must be set to RS232_MODEM.

A connection can be terminated by any of the following events. Once terminated another modem function or incoming call can take control of the serial port.

- Execution of the modemDialEnd function.
- Execution of the modemAbort or modemAbortAll functions.
- The remote device hangs up the phone line.
- An accidental loss of carrier occurs due to phone line problems.

Note that a pause of a few seconds is required between terminating a connection and initiating a new call. This pause allows the external modem time to hang up.

The reservation identifier is valid until the call is terminated and another modem function or an incoming call takes control of the port. The **modemDialEnd** function returns a DE_NotInControl error code, if another modem function or incoming call is in control of the port.

Do not call this function in a task exit handler. Use modemAbort instead.

modemDialStatus

Return Status of Dial-up Connection

Syntax

```
#include <ctools.h>
void modemDialStatus(FILE *port, reserve_id id, enum DialError * error,
   enum DialState *state);
```

Description

The **modemDialStatus** function returns the status of a remote connection initiated by the **modemDial** function. *port* specifies the serial port where the modem is installed. *id* is the port reservation identifier returned by the **modemDial** function.

The function sets the variable pointed to by *error*. If no error occurred DE_NoError is returned. Any other value indicates an error. Refer to the **Structures and Types** section for a complete description of error codes.

The function sets the variable pointed to by *state* to the current execution state of dialing operation. The state value is not valid if the error code is DE_NotInControl. Refer to the *dialup.h* section for a complete description of state codes.

Notes

The serial port type must be set to RS232_MODEM.

The reservation identifier is valid until the call is terminated and another modem function or an incoming call takes control of the port. The **modemDialStatus** function will return a DE_NotInControl error code, if another dial function or incoming call is now in control of the port.

Do not call this function in a task exit handler.

modemInit

Initialize Dial-up Modem

Syntax

```
#include <ctools.h>
enum DialError modemInit(struct ModemInit *configuration, reserve_id
   *id);
```

Description

The **modemInit** function sends an initialization string to an external dial-up modem. It is typically used to set up a modem to answer incoming calls. One modemInit function may be active on each serial port. The modemInit function handles all port sharing and multiple dialing attempts.

The ModemInit structure pointed to by *configuration* defines the serial port and modem initialization string. Refer to the **Structures and Types** section for a description of the fields in the *ModemInit* structure.

The *id* variable is set to a reservation identifier for the serial port. The identifier ensures that no other modem control function can access the serial port. This parameter must be supplied to the modemInitEnd and modemInitStatus functions.

The function returns an error code. DE_NoError indicates that the initialize operation has begun. Any other code indicates an error. Refer to the **Structures and Types** section for a complete description of error codes.

Notes

The serial port type must be set to RS232_MODEM.

The **modemInitStatus** function returns the status of the connection attempt initiated by modemInit.

The **modemInitEnd** function terminates initialization of the modem.

If a communication protocol is active on the serial port, the protocol will be disabled until the initialization is complete then re-enabled. This allows the controller to communicate with the external modem on the port. The protocol settings will also be restored when initialization is terminated with the **modemInitEnd** function.

If a **modemDial** function or an incoming call is active on the port, the **modemInit** function cannot access the port and will return an error code of DE_NotInControl.

The reservation identifier is valid until the call is terminated and another modem function or an incoming call takes control of the port.

To optimize performance, minimize the length of messages on com3. Examples of recommended uses for com3 are for local operator display terminals, and for programming and diagnostics using the ISaGRAF program.

Do not call this function in a task exit handler.

Example

Refer to the example in the Modem Initialization Example section.

modemInitEnd

Abort Initialization of Dial-up Modem

Syntax

```
#include <ctools.h>
void modemInitEnd(FILE *port, reserve_id id, enum DialError *error);
```

Description

The **modemInitEnd** function terminates a modem initialization in progress. *port* specifies the serial port where the modem is installed. *id* is the port reservation identifier returned by the **modemInit** function.

The function sets the variable pointed to by *error*. If no error occurred DE_NoError is returned. Any other value indicates an error. Refer to the *dialup.h* section for a complete description of error codes.

Notes

The serial port type must be set to RS232_MODEM.

Normally this function should be called once the **modemInitStatus** function indicates the initialization is complete.

The reservation identifier is valid until the initialization is complete or terminated, and another modem function or an incoming call takes control of the port. The **modemInitEnd** function returns a DE_NotInControl error code, if another modem function or incoming call is in control of the port.

Do not call this function in a task exit handler. Use modemAbort instead.

modemInitStatus

Return Status of Dial-up Modem Initialization

Syntax

```
#include <ctools.h>
void modemInitStatus(FILE *port, reserve_id id, enum DialError *error,
   enum DialState *state);
```

Description

The **modemInitStatus** function returns the status of a modem initialization started by the **modemInit** function. *port* specifies the serial port where the modem is installed. *id* is the port reservation identifier returned by the **modemInit** function.

The function sets the variable pointed to by *error*. If no error occurred DE_NoError is returned. Any other value indicates an error. Refer to the **Structures and Types** section for a complete description of error codes.

The function sets the variable pointed to by *state* to the current execution state of the dialing operation. The state value is not valid if the error code is DE_NotInControl. Refer to the *dialup.h* section for a complete description of state codes.

Notes

The serial port type must be set to RS232_MODEM.

The port will remain in the DS_Calling state until modem initialization is complete or fails. The application should wait until the state is not DS_Calling before calling the **modemInitEnd** function.

The reservation identifier is valid until the initialization is complete or terminated, and another modem function or an incoming call takes control of the port.

Do not call this function in a task exit handler.

modemNotification

Notify the modem handler of an important event

Syntax

```
#include <ctools.h>
void modemNotification(UINT16 port_index);
```

Description

The `modemNotification` function notifies the dial-up modem handler that an interesting event has occurred. This informs the modem handler not to disconnect an incoming call when an outgoing call is requested with `modemDial`.

This function is used with custom communication protocols. The function is usually called when a message is received by the protocol, although it can be called for other reasons.

The `port_index` indicates the serial port that received the message.

Notes

The serial port type must be set to RS232_MODEM.

The dial-up connection handler prevents outgoing calls from using the serial port when an incoming call is in progress and communication is active. If communication stops for more than five minutes, then outgoing call requests are allowed to end the incoming call. This prevents problems with the modem or the calling application from permanently disabling outgoing calls.

The function is used with programs that dial out through an external modem using the `modemDial` function. It is not required where the modem is used for dialing into the controller only.

mTcpGetConfig

Get Modbus/TCP Protocol Settings

Syntax

```
#include <ctools.h>
UINT16 mTcpGetConfig(MTCP_CONFIGURATION * pSettings)
```

Description

The **mTcpGetConfig** function copies the Modbus/TCP protocol settings to the structure pointed to by *pSettings*. The structure MTCP_CONFIGURATION is described in the *Structures and Types* section.

The settings are common to all connections using the Modbus/TCP protocol. If the Modbus/TCP server is currently running, 1 is returned. If the server is not running, 0 is returned.

mTcpGetInterface

Get Modbus IP Interface Settings

Syntax

```
#include <ctools.h>
BOOLEAN mTcpGetInterface( COM_INTERFACE ifType, MTCP_IF_SETTINGS *
pSettings );
```

Description

The **mTcpGetInterface** function is used to obtain the interface settings for Modbus IP protocols on the specified interface. If the selected interface is invalid, FALSE is returned; otherwise TRUE is returned and the settings are copied to the structure pointed to by *pSettings*.

The valid value for *ifType* is CIF_Ethernet1. The enumeration type COM_INTERFACE and the structure MTCP_IF_SETTINGS are described in the *Structures and Types* section.

mTcpGetInterfaceEx

Get Modbus IP Interface Extended Settings

Syntax

```
#include <ctools.h>
BOOLEAN mTcpGetInterfaceEx(
      COM_INTERFACE ifType,
      MTCP_IF_SETTINGS_EX * pSettings
      );
```

Description

This function returns the interface settings used for Modbus IP protocols, including Enron Modbus settings.

The function has two parameters:

- `ifType` specifies the interface. The valid value is `CIF_Ethernet1`.
- `pSettings` is a pointer to a Modbus IP interface extended settings structure. The settings are copied to this structure.

The function returns `TRUE` if the specified interface is valid and `FALSE` otherwise. The enumeration type COM_INTERFACE and the structure MTCP_IF_SETTINGS_EX are described in the *Structures and Types* section.

mTcpGetProtocol

Get Modbus IP Protocol Settings

Syntax

```
#include <ctools.h>
BOOLEAN mTcpGetProtocol(IP_PROTOCOL_TYPE type, IP_PROTOCOL_SETTINGS *
   pSettings);
```

Description

The **mTcpGetProtocol** function copies the settings for a specific Modbus IP or DNP IP protocol to the structure pointed to by *pSettings*. The protocol type is selected with the *type* argument and it may be set to any of the following: IPP_ModbusTcp, IPP_ModbusRtuOverUdp, IPP_ModbusAsciiOverUdp, IPP_DnpOverTcp or IPP_DnpOverUdp.

If the protocol type is valid, the settings are copied and TRUE is returned. If the protocol type is invalid, FALSE is returned and nothing is copied.

The structure IP_PROTOCOL_SETTINGS is described in the *Structures and Types* section.

See Also

mTcpSetProtocol, **mTcpGetInterfaceEx**

mTcpSetConfig

Set Modbus/TCP Protocol Settings

Syntax

```
#include <ctools.h>
BOOLEAN mTcpSetConfig(MTCP_CONFIGURATION * pSettings);
```

Description

The **mTcpSetConfig** function is used to configure settings common to all connections using the Modbus/TCP protocol. All existing connections are maintained after calling this function. For this reason it is recommended that all connections using this protocol be closed before calling this function.

If this function is used to change the port number or maximum number of server connections, then the Modbus/TCP Server task is ended and re-started with the new settings. Port number changes will only affect new connections made after calling this function. All other changes take effect on existing as well as new connections.

The function copies settings from the structure pointed to by *pSettings* to the Modbus/TCP protocol configuration and returns TRUE. The structure MTCP_CONFIGURATION is described in the *Structures and Types* section. If there is an invalid setting, FALSE is returned and the settings are not copied.

To save these settings with the controller settings in flash memory so that they are loaded on controller reset, call **flashSettingsSave** as shown below.

```
request_resource(FLASH_MEMORY);
flashSettingsSave(CS_RUN);
       release_resource(FLASH_MEMORY);
```

mTcpSetInterface

Set Modbus IP Interface Settings

Syntax

```
#include <ctools.h>
BOOLEAN mTcpSetInterface( COM_INTERFACE ifType, MTCP_IF_SETTINGS *
pSettings );
```

Description

The **mTcpSetInterface** function is used to set the interface settings used by the Modbus IP protocols. If the selected interface or the settings are invalid, FALSE is returned; otherwise TRUE is returned and the settings are set for the specified interface.

The valid value for *ifType* is CIF_Ethernet1. The enumeration type COM_INTERFACE and the structure MTCP_IF_SETTINGS are described in the *Structures and Types* section.

To save these settings with the controller settings in flash memory so that they are loaded on controller reset, call **flashSettingsSave** as shown below.

```
request_resource(FLASH_MEMORY);
flashSettingsSave(CS_RUN);
       release_resource(FLASH_MEMORY);
```

mTcpSetInterfaceEx

Set Modbus IP Interface Extended Settings

Syntax

```
#include <ctools.h>
BOOLEAN mTcpSetInterfaceEx(
      COM_INTERFACE ifType,
      MTCP_IF_SETTINGS_EX * pSettings
      );
```

Description

This function sets the interface settings used for Modbus IP protocols, including Enron Modbus settings.

The function has two parameters:

- `ifType` specifies the interface. The valid value is `CIF_Ethernet1`.
- `pSettings` is a pointer to a Modbus IP interface extended settings structure that contains the desired settings.

The function returns `TRUE` if the specified interface and settings are valid and `FALSE` otherwise.

To save these settings with the controller settings in flash memory so that they are loaded on controller reset, call **flashSettingsSave** as shown below.

```
request_resource(FLASH_MEMORY);
flashSettingsSave(CS_RUN);
release_resource(FLASH_MEMORY);
```

Notes

The IO_SYSTEM resource must be requested before calling this function with TelePACE firmware.

The settings take effect for all new connections made thereafter on the specified interface. Existing connections are not affected.

mTcpSetProtocol

Set Modbus IP Protocol Settings

Syntax

```
#include <ctools.h>
BOOLEAN mTcpSetProtocol(IP_PROTOCOL_TYPE type, IP_PROTOCOL_SETTINGS *
   pSettings);
```

Description

The **mTcpSetProtocol** function is used to configure settings for a specific Modbus IP protocol. The protocol type argument may be set to any of the following: IPP_ModbusTcp, IPP_ModbusRtuOverUdp, IPP_ModbusAsciiOverUdp, IPP_DnpOverTcp or IPP_DnpOverUdp.

If this function is used to change the port number, then the server task for the selected protocol is ended and re-started with the new settings. Port number changes will only affect new connections made after calling this function. All other changes take effect on existing as well as new connections.

This function may be used to change the server enable status. The *serverEnabled* setting selects whether the server is enabled for the selected protocol. If this flag is set to TRUE the controller supports incoming slave messages that use the selected protocol. Setting this flag to FALSE prevents the controller from processing slave messages for this protocol. Master messaging is always enabled.

The function copies the settings from the structure pointed to by *pSettings* to the settings of the specified protocol and returns TRUE. The structure IP_PROTOCOL_SETTINGS is described in the *Structures and Types* section. If there is an invalid setting, FALSE is returned and the settings are not copied.

To save these settings with the controller settings in flash memory so that they are loaded on controller reset, call **flashSettingsSave** as shown below.

```
request_resource(FLASH_MEMORY);
flashSettingsSave(CS_RUN);
release_resource(FLASH_MEMORY);
```

Notes

The IO_SYSTEM resource must be requested before calling this function with TelePACE firmware.

See Also

mTcpGetProtocol, **mTcpSetInterfaceEx**

mTcpMasterClose

Close Modbus IP Master Messaging Session

Syntax

```
#include <ctools.h>
BOOLEAN mTcpMasterClose( UINT32 connectID );
```

Description

The **mTcpMasterClose** function returns the specified *connectID* to the pool of available connections so that it may be re-used for other new connections. FALSE is returned if the specified *connectID* is invalid, or if the connection has not been disconnected; otherwise TRUE is returned and the *connectID* is released.

After calling this function, the function **mTcpMasterStatus** may no longer be called with this connectID.

The function **mTcpMasterDisconnect** must be called first before calling **mTcpMasterClose** to disconnect and end the mastering task. If this is not done, **mTcpMasterClose** returns FALSE and the *connectID* is not released.

Example

See example for Master Message Example Using mTcpMasterMessage.

mTcpMasterDisconnect

Disconnect Modbus IP Master Connection

Syntax

```
#include <ctools.h>
BOOLEAN mTcpMasterDisconnect( UINT32 connectID );
```

Description

The **mTcpMasterDisconnect** function signals the mastering task to tell it to disconnect from the remote slave and end the task. FALSE is returned if the specified *connectID* is invalid; otherwise a TRUE is returned.

FALSE is also returned if the master task has not completed the last command. In this case, the **mTcpMasterDisconnect** function must be called repeatedly until TRUE is returned.

After calling the **mTcpMasterDisconnect** function, the function **mTcpMasterStatus** may be used to determine the progress of the disconnect. These functions may not be called after calling the function **mTcpMasterClose** with the same *connectID*. The results of such a call are unpredictable, as the *connectID* may have been re-used already for a new connection.

After calling **mTcpMasterDisconnect** successfully, call **mTcpMasterClose** to return the connection ID to the pool of available connections.

Example

See the example in the Example Programs chapter under the section **Master IP Message Example**.

mTcpMasterMessage

Send a Modbus IP Master Message

Syntax

```
#include <ctools.h>
MODBUS_CMD_STATUS mTcpMasterMessage( UINT32 connectID, IP_ADDRESS
remoteIP, IP_PROTOCOL_TYPE protocolType, UINT16 function, UINT16
slaveStation, UINT16 slaveRegister, UINT16 masterRegister, UINT16 length,
UINT16 timeout);
```

Description

The **mTcpMasterMessage** function builds a Modbus command message using the specified Modbus IP protocol and signals the mastering task to tell it to send the command.

The *connectID* specifies the connection ID returned by the function **mTcpMasterOpen** which was called to create a mastering task to service this connection.

The *remoteIP* specifies the IP address of the remote slave. The value of *remoteIP* may be the same or different from the IP address used in **mTcpMasterOpen** or in a previous call to **mTcpMasterMessage**. This is possible because the *connectID* represents the allocation of a connection from the connection pool and may be used to connect to any IP address.

When the IP address is changed between function calls, the current connection is closed and a connection to the new IP address is automatically established. Note that it is more efficient to allocate one connectID and its associated master task for each remoteIP because the connection remains connected to one IP address. However, if there are fewer connections available than there are remote slaves, the same connectID can be used to re-connect to multiple IP addresses.

Valid values for *protocolType* are*:* IPP_ModbusTcp, IPP_ModbusRtuOverUdp, or IPP_ModbusAsciiOverUdp.

The remaining arguments are used in the same way as they are used in **master_message** to send a serial Modbus command:

- *function* specifies the Modbus function code. Refer to the communication protocol manual for supported function codes.
- *slaveStation* specifies the network address of the slave station. This is also known as the slave station number.
- *slaveRegister* specifies a Modbus register in the slave station. Depending on the protocol function code, data may be read or written at this location.
- *masterRegister* specifies a Modbus register in the master (this controller). Depending on the protocol function code, data may be read or written at this location.
- *length* specifies the number of registers.

The *timeout*, in tenths of seconds, tells the mastering task how long to wait for a response from the slave. For TCP protocols the same *timeout* is also used by the mastering task as the time to wait for a connection to be re-established if this is required. To disable the timeout and have the mastering task wait forever for a response or a connection to be established, set the *timeout* to 0. Note that this *timeout* replaces the initial timeout specified in **mTcpMasterOpen**. This allows **mTcpMasterMessage** to specify different timeout values for different IP addresses each time the function is called.

If a TCP protocol connection is left idle and the master idle timeout occurs, the connection is closed to conserve resources at the remote slave. The connection is automatically re-established the next time **mTcpMasterMessage** is called. Master idle timeout is set using the function **mTcpSetProtocol**. Note that closing the TCP/IP connection in an idle timeout does not return the connection ID to the pool of available connections. The connection ID remains allocated to this master session until **mTcpMasterClose** is called.

An error code is returned if the specified *connectID* is invalid, or if a command argument is invalid; otherwise MM_SENT is returned. If the last command message is still in progress, the command status is returned and a new message is not sent. Note that the **mTcpMasterMessage** always returns immediately. It is the mastering task created in the background that services the IP connection.

The command status returned by this function is set to MM_SENT if a valid master message was sent. Other values returned for the command status are described for the enumeration type MODBUS_CMD_STATUS in the *Structures and Types* section. Use the function **mTcpMasterStatus** to determine the progress of the Modbus IP command and the slave response. The command status will be set to MM_RECEIVED when the response to the message is received.

Notes

Refer to the communication protocol manual for more information.

The IO_SYSTEM resource must be requested before calling this function.

Example

See the example in the Example Programs chapter under the section **Master IP Message Example**.

mTcpMasterOpen

Open a Modbus IP Master Connection

Syntax

```
#include <ctools.h>
BOOLEAN mTcpMasterOpen( IP_ADDRESS remoteIP, IP_PROTOCOL_TYPE
protocolType, CONNECTION_TYPE appType, UINT16 timeout, UINT32 *
connectID, MODBUS_CMD_STATUS * cmdStatus );
```

Description

The **mTcpMasterOpen** function allocates the resources needed to make a Modbus IP master connection to a remote IP address. These resources consist of a connection ID from the connection pool and the creation of a task to service the master IP connection. When the task is created an initial connection to *remoteIP* is attempted. However, the connection ID and master task are not restricted to just one *remoteIP.* The currently connected IP address may be disconnected and connected to a different IP address any time **mTcpMasterMessage** is called with a different *remoteIP* for this connection ID. See **mTcpMasterMessage** for more details.

Valid values for *protocolType* are*:* IPP_ModbusTcp, IPP_ModbusRtuOverUdp, or IPP_ModbusAsciiOverUdp. There is only one valid value for *appType*: CT_MasterCApp.

For TCP protocols, the *timeout* specifies the time, in tenths of seconds, to wait for a connection to be established whenever a connection is attempted by the created master task. To disable the timeout and wait forever for a connection to be established, set the *timeout* to 0.

Each time this function is called a new connection ID is allocated from the connection pool. If the number of currently allocated connections is less than 20, a task is created to service the allocated connection and the function returns TRUE. If there are no connections available, or if there is an error in one of the arguments, FALSE is returned and an error code is copied to the value pointed by *cmdStatus*.

The new mastering task establishes the initial connection and sends Modbus IP master messages each time **mTcpMasterMessage** is called. Use the function **mTcpMasterStatus** to determine the status of the connection or master message in progress.

The connection ID for this master connection is copied to the value pointed to by *connectID*. This ID must be used when calling the remaining master messaging API functions for this connection: **mTcpMasterMessage, mTcpMasterStatus, mTcpMasterDisconnect,** and **mTcpMasterClose**

The enumeration types and structures used for the function arguments are described in the *Structures and Types* section.

Notes

The functions **mTcpMasterDisconnect** and **mTcpMasterClose** must be called to disconnect and return this connection ID to the pool of available connections. Even if the connection to the remote IP is disconnected, manually or automatically after an idle timeout, the connection ID remains allocated until **mTcpMasterDisconnect** is called to disconnect and end the mastering task, and **mTcpMasterClose** is called to return the connection ID.

Note that there are only 20 connections available for all Modbus IP master and slave connections. Use the function **ipGetConnectionSummary** obtain the number of master and slave connections that are currently active.

If the initial connection started by this function fails, the connection will be attempted again if necessary each time **mTcpMasterMessage** is called.

See the function **mTcpMasterMessage** for a discussion of whether to allocate one or several connections when polling multiple remote IP addresses.

Example

See example for Master Message Example Using mTcpMasterMessage**.**

mTcpMasterStatus

Modbus IP Master Command Status

Syntax

```
#include <ctools.h>
BOOLEAN mTcpMasterStatus( UINT32 connectID, MODBUS_CMD_STATUS * cmdStatus
);
```

Description

The **mTcpMasterStatus** function obtains the Modbus command status for the connection specified by *connectID.*

This function copies the master command status to the value pointed to by *cmdStatus*. FALSE is returned if the specified *connectID* is invalid; otherwise TRUE is returned and the status is copied.

This function may not be called after calling the function **mTcpMasterClose** with the same *connectID*. The results of such a call are unpredictable, as the *connectID* may have been re-used already for a new connection.

Expected values returned for the command status are described for the enumeration type MODBUS_CMD_STATUS in the *Structures and Types* section.

Example

See example for Master Message Example Using mTcpMasterMessage**.**

mTcpRunServer

Run Modbus IP Servers

Syntax

```
#include <ctools.h>
void mTcpRunServer( BOOLEAN state );
```

Description

The **mTcpRunServer** function is used to start the servers for each IP protocol. The IP protocols include Modbus/TCP, Modbus RTU over UDP, Modbus ASCII over UDP, DNP over TCP, and DNP over UDP.

Calling this function with TRUE starts the servers according to the IP protocol settings: If the server enabled setting for the protocol is TRUE, then the server is started. If the server enabled setting for the protocol is FALSE, then the server is stopped. Calling this function with FALSE stops each IP protocol server and updates IP protocol settings accordingly.

Use the function **mTcpSetProtocol** to enable or disable a server for a specific IP protocol.

This function should only be needed in the context of the startup function **appstart**.

ntohl

Syntax

```
#include <ctools.h>
unsigned long ntohl
(
       unsigned long longValue
);
```

Function Description

This function converts a long value from network byte order to host byte order.

Parameters

longValue The value to convert

Returns

The converted value.

ntohs

Syntax

```
#include <ctools.h>
unsigned short ntohs
(
       unsigned short shortValue
);
```

Function Description

This function converts a short value from network byte order to host byte order.

Parameters

shortValue	The value to convert

Returns

The converted value.

overrideDbase

Overwrite Value in Forced I/O Database (TelePACE firmware only)

Syntax

```
#include <ctools.h>
BOOLEAN overrideDbase(UINT16 type, UINT16 address, INT16 value);
```

Description

The **overrideDbase** function writes *value* to the I/O database even if the database register is currently forced. *type* specifies the method of addressing the database. *address* specifies the location in the database.

If the register is currently forced, the register remains forced but forced to the new *value.*

If the *address* or addressing *type* is not valid, the I/O database is left unchanged and FALSE is returned; otherwise TRUE is returned. The table below shows the valid address types and ranges.

Type	Address Ranges	Register Size
MODBUS	00001 to NUMCOIL 10001 to 10000 + NUMSTATUS 30001 to 30000 + NUMINPUT 40001 to 40000 + NUMHOLDING	1 bit 1 bit 16 bit 16 bit
LINEAR	0 to NUMLINEAR-1	16 bit

Notes

When writing to LINEAR digital addresses, *value* is a bit mask, which writes data to 16 1-bit registers at once.

The I/O database is not modified when the controller is reset. It is a permanent storage area, which is maintained during power outages.

Refer to the **Functions Overview** chapter for more information.

The IO_SYSTEM resource must be requested before calling this function.

See Also

Example

```
#include <ctools.h>
int main(void)
{
       request_resource(IO_SYSTEM);

       overrideDbase(MODBUS, 40001, 102);
       overrideDbase(LINEAR, 302, 330);

       release_resource(IO_SYSTEM);
}
```

pidExecute

Execute PID control algorithm

Syntax

```
#include <ctools.h>
BOOLEAN pidExecute(PID_DATA * pData);
```

Description

This function executes the PID algorithm. The function may be called as often as desired, but must be called at least once per the value in the period field for proper operation.

The function has one parameter. *pData* is a pointer to a structure containing the PID block data and outputs.

The function returns TRUE if the PID block executed. The function returns FALSE if it was not time for execution.

Notes

To properly initialize the PID algorithm do one of the following.

- Call the **pidInitialize** function once before calling this function the first time, or
- put the PID algorithm in manual mode (`autoMode = FALSE` in `PID_DATA`) for the first call to the **pidExecute** function.

Example

This example initializes one PID control structure and executes the control algorithm continuously. Input data is read from analog inputs. Output data is written to analog outputs.

```
#include <ctools.h>

// event number to signal when I/O scan completes
#define IO_COMPLETE 0

int main(void)
{
      INT16 ainData[4];      // analog input data
      INT16 aoutData[4];     // analog output data
      PID_DATA pidData;      // PID algorithm data
      BOOLEAN executed;      // indicates if PID executed

      // read analog input
      ioRequest(MT_Ain4, 0);
      ioNotification(IO_COMPLETE);
      wait_event(IO_COMPLETE);
      ioReadAin4(0, ainData);

      // get initial process value from analog input
      pidData.pv = ainData[0];

      // configure PID block
      pidData.sp          = 1000;
      pidData.gain        = 1;
      pidData.reset       = 100;
```

```
        pidData.rate         = 0;
        pidData.deadband     = 10;
        pidData.fullScale    = 32767;
        pidData.zeroScale    = 0;
        pidData.manualOutput = 0;
        pidData.period       = 1000;
        pidData.autoMode     = TRUE;

        // initialize the PID block
        pidInitialize(&pidData);

        // main loop
        while (TRUE)
        {
           // execute all I/O requests
           ioRequest(MT_Ain4, 0);
           ioNotification(IO_COMPLETE);
           wait_event(IO_COMPLETE);

           // get process input
           ioReadAin4(0, ainData);
           pidData.pv = ainData[0];

           // execute the PID block
           executed = pidExecute(&pidData);

           // if the output changed
           if (executed)
           {
              // write the output to analog output module
              aoutData[0] = pidData.output;
              ioWriteAout4(0, aoutData);
              ioRequest(MT_Aout4, 0);
           }

           // release processor to other priority 254 tasks
           release_processor();
        }
}
```

pidInitialize

Initialize PID controller data

Syntax

```
#include <ctools.h>
void pidInitialize(PID_DATA * pData);
```

Description

This function initializes the PID algorithm data.

The function has one parameter. *pData* is a pointer to a structure containing the PID data and outputs.

The function should be called once before calling the **pidExecute** function for the first time. The structure pointed to by *pData* must contain valid values for sp, pv, and manualOutput before calling the function.

The function has no return value.

See Also

pidExecute

Example

See the example for pidExecute.

pollABSlave

Poll DF1 Slave for Response

Syntax

```
#include <ctools.h>
UINT16 pollABSlave(FILE *stream, UINT16 slave);
```

Description

The **pollABSlave** function is used to send a poll command to the slave station specified by *slave* in the DF1 Half Duplex protocol configured for the specified port. *stream* specifies the serial port.

The function returns **FALSE** if the slave number is invalid, or if the protocol currently installed on the specified serial port is not an DF1 Half Duplex protocol. Otherwise it returns **TRUE** and the protocol command status is set to **MM_SENT**.

Notes

See the example in the Example Programs chapter under the section **Master Message Example Using DF1 Protocol.** The **pollABSlave** function is used in the sample polling function "poll_for_response" shown in this example.

See Also

resetAllABSlaves

Example

This program segment polls slave station 9 for a response communicating on the **com2** serial port.

```
#include <ctools.h>

pollABSlave(com2, 9);
```

poll_event

Test for Event Occurrence

Syntax

```
#include <ctools.h>
BOOLEAN poll_event(UINT32 event);
```

Description

The **poll_event** function tests if an event has occurred.

The **poll_event** function returns **TRUE,** and the event counter is decrements, if the event has occurred. Otherwise it returns **FALSE**.

The current task always continues to execute.

Notes

Refer to the **Real Time Operating System** section for more information on events.

Valid events are numbered 0 to RTOS_EVENTS - 1. Any events defined in primitiv.h are not valid events for use in an application program.

Example

This program implements a somewhat inefficient transfer of data between **com1** and **com2**. (It would be more efficient to test for EOF from getc).

```
#include <ctools.h>

int main(void)
{
       while(TRUE)
       {
          if (poll_event(COM1_RCVR))
             fputc(getc(com1), com2);
          if (poll_event(COM2_RCVR))
             fputc(getc(com2), com1);

          /* Allow other tasks to execute */
          release_processor();
       }
}
```

poll_message

Test for Received Message

Syntax

```
#include <ctools.h>
envelope *poll_message(void);
```

Description

The **poll_message** function tests if a message has been received by the current task.

The **poll_message** function returns a pointer to an envelope if a message has been received. It returns **NULL** if no message has been received.

The current task always continues to execute.

Notes

Refer to the **Real Time Operating System** section for more information on messages.

See Also

poll_event

Example

This task performs a function continuously, and processes received messages (from higher priority tasks) when they are received.

```
#include <ctools.h>

void task(void)
{
       envelope *letter;

       while(TRUE)
       {
           letter=poll_message();
           if (letter != NULL)
              /* process the message now */

           /* more code here */
       }
}
```

poll_resource

Test Resource Availability

Syntax

```
#include <ctools.h>
BOOLEAN poll_resource(UINT32 resource);
```

Description

The **poll_resource** function tests if the resource specified by *resource* is available. If the resource is available it is given to the task.

The **poll_resource** function returns **TRUE** if the resource is available. It returns **FALSE** if it is not available.

The current task always continues to execute.

Notes

Refer to the **Real Time Operating System** section for more information on resources.

See Also

poll_event, poll_message

portIndex

Get Index of Serial Port

Syntax

```
#include <ctools.h>
UINT16 portIndex(FILE *stream);
```

Description

The **portIndex** function returns an array index for the serial port specified by *stream*. It is guaranteed to return a value suitable for an array index, in increasing order of external serial port numbers, if no error occurs.

If the stream is not recognized, SERIAL_PORTS is returned, to indicate an error.

See Also

portStream

portStream

Get Serial Port Corresponding to Index

Syntax

```
#include <ctools.h>
FILE *portStream(UINT16 index);
```

Description

The **portStream** function returns the file pointer corresponding to *index*. This function is the inverse of the **portIndex** function. If the index is not valid, the NULL pointer is returned.

See Also

portIndex

queue_mode

Control Serial Data Transmission

Syntax

```
#include <ctools.h>
void queue_mode(UCHAR port, INT16 mode);
```

Description

The **queue_mode** function controls transmission of the serial data. Normally data output to a serial port are placed in the transmit buffer and transmitted as soon as the hardware is ready. If queuing is enabled, the characters are held in the transmit buffer until queuing is disabled. If the buffer fills, queuing is disabled automatically.

port specifies the serial port. If it is not valid the function has no effect.

mode specifies the queuing control. It may be **DISABLE** or **ENABLE**.

Notes

Queuing is most often used with communication protocols that use character timing for message framing. Its uses in an application program are limited.

readBoolVariable

Read ISaGRAF Boolean Variable (ISaGRAF firmware only)

Syntax

```
#include <ctools.h>
BOOLEAN readBoolVariable(UCHAR * varName, UCHAR * value)
```

Description

This function returns the current value of the specified boolean variable.

The variable is specified by its name expressed as a character string. The name is case insensitive (The ISaGRAF Dictionary also treats variable names as case insensitive). If the variable is found, TRUE is returned and the variable value is written to the unsigned char value pointed to by *value*. If the variable is not found or if the ISaGRAF Symbols Status is invalid, FALSE is returned and the current value is left unchanged. The ISaGRAF Symbols Status is invalid if the Application TIC code download and Application Symbols download do not share the same symbols CRC checksum.

Notes

This function requires the ISaGRAF Application Symbols to be downloaded to the controller in addition to the Application TIC code. This function provides a convenient method to access ISaGRAF variables by name; however, because the variable name must be looked up in the ISaGRAF variable list each call, the performance of the function may be slow for large numbers of variables. For better performance, use the variable's network address and the **dbase** function.

The IO_SYSTEM system resource must be requested before calling this function.

See Also

readIntVariable, readRealVariable

Example

This program displays the contents of the boolean variable named "Switch1".

```
#include <ctools.h>

int main(void)
{
   BOOLEAN       status;
   UCHAR char value;

   request_resource(IO_SYSTEM);
   status = readBoolVariable("Switch1", &value);
   release_resource(IO_SYSTEM);

   fprintf(com1,"status = %u, Switch1 = %d\r\n", status, value);
}
```

readBattery

Read Lithium Battery Voltage

Syntax

```
#include <ctools.h>
INT16 readBattery(void);
```

Description

The **readBattery** function returns the RAM backup battery voltage in millivolts. The range is 0 to 5000 mV. A normal reading is about 3600 mV.

Example

```
#include <ctools.h>

if (readBattery() < 2500)
{
       fprintf(com1, “Battery Voltage is low\r\n”);
}
```

readIntVariable

Read ISaGRAF Integer Variable (ISaGRAF firmware only)

Syntax

```
#include <ctools.h>
BOOLEAN readIntVariable(UCHAR * varName, INT32 * value)
```

Description

This function returns the current value of the specified integer variable.

The variable is specified by its name expressed as a character string. The name is case insensitive (The ISaGRAF Dictionary also treats variable names as case insensitive). If the variable is found, TRUE is returned and the variable value is written to the signed long value pointed to by *value*. If the variable is not found or if the ISaGRAF Symbols Status is invalid, FALSE is returned and the current value is left unchanged. The ISaGRAF Symbols Status is invalid if the Application TIC code download and Application Symbols download do not share the same symbols CRC checksum.

Notes

This function requires the ISaGRAF Application Symbols to be downloaded to the controller in addition to the Application TIC code. This function provides a convenient method to access ISaGRAF variables by name; however, because the variable name must be looked up in the ISaGRAF variable list each call, the performance of the function may be slow for large numbers of variables. For better performance, use the variable's network address and the **dbase** function.

The IO_SYSTEM system resource must be requested before calling this function.

See Also

readRealVariable

Example

This program displays the contents of the integer variable named "Temperature".

```
#include <ctools.h>

int main(void)
{
       BOOLEAN   status;
       INT32  value;

       request_resource(IO_SYSTEM);
       status = readIntVariable("Temperature", &value);
       release_resource(IO_SYSTEM);

       fprintf(com1,"status = %u, Temp = %ld\r\n", status, value);
}
```

readMsgVariable

Read ISaGRAF Message Variable (ISaGRAF firmware only)

Syntax

```
#include <ctools.h>
BOOLEAN readMsgVariable(UCHAR * varName, UCHAR * msg)
```

Description

This function returns the current value of the specified message variable.

The variable is specified by its name expressed as a character string. The name is case insensitive (The ISaGRAF Dictionary also treats variable names as case insensitive). If the variable is found, TRUE is returned and the message is written to the string pointed to by *msg*. If the variable is not found or if the ISaGRAF Symbols Status is invalid, FALSE is returned and the buffer is left unchanged. The ISaGRAF Symbols Status is invalid if the Application TIC code download and Application Symbols download do not share the same symbols CRC checksum.

The pointer *msg* must point to a character string large enough to hold the maximum length declared for the specified message variable plus two length bytes and a null termination byte (i.e. max declared length + 3). ISaGRAF message variables have the following format:

Byte Location	Description
0	Maximum length as declared in ISaGRAF Dictionary (1 to 255)
1	Current Length = number of bytes up to first null byte in message data (0 to maximum length)
2	First message data byte
…	
max + 1	Last byte in message buffer
max + 2	Null termination byte (Terminates a message having the maximum length.)

Notes

This function requires the ISaGRAF Application Symbols to be downloaded to the controller in addition to the Application TIC code. This function provides a convenient method to access ISaGRAF variables by name; however, because the variable name must be looked up in the ISaGRAF variable list each call, the performance of the function may be slow for large numbers of variables. For better performance, use the variable's network address and the **dbase** function.

The IO_SYSTEM system resource must be requested before calling this function.

See Also

readIntVariable, readRealVariable

Example

This program displays the contents of the message variable named “msgData” of maximum length 20.

```
#include <ctools.h>

int main(void)
{
          BOOLEAN status;
          UCHAR msg[23];

          request_resource(IO_SYSTEM);
          status = readMsgVariable("msgData", msg);
          release_resource(IO_SYSTEM);

          fprintf(com1,"status = %u, max length = %d, current length =
%d,
          message = %s\r\n", status, msg[0], msg[1], msg + 2);
}
```

readRealVariable

Read ISaGRAF Real Variable (ISaGRAF firmware only)

Syntax

```
#include <ctools.h>
BOOLEAN readRealVariable(UCHAR * varName, float * value)
```

Description

This function returns the current value of the specified real (i.e. floating point) variable.

The variable is specified by its name expressed as a character string. The name is case insensitive (The ISaGRAF Dictionary also treats variable names as case insensitive). If the variable is found, TRUE is returned and the variable value is written to the floating point value pointed to by *value*. If the variable is not found or if the ISaGRAF Symbols Status is invalid, FALSE is returned and the current value is left unchanged. The ISaGRAF Symbols Status is invalid if the Application TIC code download and Application Symbols download do not share the same symbols CRC checksum.

Notes

This function requires the ISaGRAF Application Symbols to be downloaded to the controller in addition to the Application TIC code. This function provides a convenient method to access ISaGRAF variables by name; however, because the variable name must be looked up in the ISaGRAF variable list each call, the performance of the function may be slow for large numbers of variables. For better performance, use the variable's network address and the **dbase** function.

The IO_SYSTEM system resource must be requested before calling this function.

See Also

readIntVariable

Example

This program displays the contents of the real variable named "Flow".

```
#include <ctools.h>

int main(void)
{
       BOOLEAN   status;
       float     value;

       request_resource(IO_SYSTEM);
       status = readRealVariable("Flow", &value);
       release_resource(IO_SYSTEM);

       fprintf(com1,"status = %u, Flow = %f\r\n", status, value);
}
```

readStopwatch

Read Stopwatch Timer

Syntax

```
#include <ctools.h>
UINT32 readStopwatch(void)
```

Description

The **readStopwatch** function reads the stopwatch timer. The stopwatch time is in ms and has a resolution of 10 ms. The stopwatch time rolls over to 0 when it reaches the maximum value for an unsigned long integer: 4,294,967,295 ms (or about 49.7 days).

Example

This program measures the execution time in ms of an operation.

```
#include <ctools.h>

int main(void)
{
          UINT32 startTime, endTime;

          startTime = readStopwatch();
          /* operation to be timed */
          endTime = readStopwatch();

          fprintf(com1,"Execution time = %lu ms\r\n", endTime -
startTime);
}
```

readThermistor

Read Controller Ambient Temperature

Syntax

```
#include <ctools.h>
INT16 readThermistor(UINT16 scale);
```

Description

The **readThermistor** function returns the temperature measured at the main board in the specified temperature *scale*. If the temperature scale is not recognized, the temperature is returned in Celsius. The *scale* may be T_CELSIUS, T_FAHRENHEIT, T_KELVIN or T_RANKINE.

The temperature is rounded to the nearest degree.

Example

```
#include <ctools.h>

void checkTemperature(void)
{
          INT16 temperature;

          temperature = readThermistor(T_FAHREHEIT);
          if (temperature < 0)
             fprintf(com1, “It’s COLD!!!\r\n”);
          else if (temperature > 90)
             fprintf(com1, “It’s HOT!!!\r\n”);
}
```

readTimerVariable

Read ISaGRAF Timer Variable (ISaGRAF firmware only)

Syntax

```
#include <ctools.h>
BOOLEAN readTimerVariable(unsigned char * varName, unsigned long * value)
```

Description

This function returns the current value in milliseconds of the specified timer variable. The maximum value returned is 86399999 ms (or 24 hours). The specified timer may be active or stopped.

The variable is specified by its name expressed as a character string. The name is case insensitive (The ISaGRAF Dictionary also treats variable names as case insensitive). If the variable is found, TRUE is returned and the variable value is written to the unsigned long value pointed to by *value*. If the variable is not found or if the ISaGRAF Symbols Status is invalid, FALSE is returned and the current value is left unchanged. The ISaGRAF Symbols Status is invalid if the Application TIC code download and Application Symbols download do not share the same symbols CRC checksum.

Notes

This function requires the ISaGRAF Application Symbols to be downloaded to the controller in addition to the Application TIC code. This function provides a convenient method to access ISaGRAF variables by name; however, because the variable name must be looked up in the ISaGRAF variable list each call, the performance of the function may be slow for large numbers of variables. For better performance, use the variable's network address and the **dbase** function.

The IO_SYSTEM system resource must be requested before calling this function.

See Also

readIntVariable, readRealVariable

Example

This program displays the contents of the timer variable named "Time1".

```
#include <ctools.h>

int main(void)
{
       BOOLEAN status;
       UINT32 value;

       request_resource(IO_SYSTEM);
       status = readTimerVariable("Time1", &value);
       release_resource(IO_SYSTEM);

       fprintf(com1,"status = %u, Time1 = %lu\r\n", status, value);
}
```

receive_message

Receive a Message

Syntax

```
#include <ctools.h>
envelope *receive_message(void);
```

Description

The **receive_message** function reads the next available envelope from the message queue for the current task. If the queue is empty, the task is blocked until a message is sent to it.

The **receive_message** function returns a pointer to an `envelope` structure.

Notes

Refer to the **Real Time Operating System** section for more information on messages.

See Also

send_message

Example

This task waits for messages, then prints their contents. The envelopes received are returned to the operating system.

```
#include <ctools.h>

void show_message(void)
{
       envelope *msg;
       while (TRUE)
       {
          msg = receive_message();
          fprintf(com1,"Message data %ld\r\n", msg->data);
          deallocate_envelope(msg);
       }
}
```

recv

Syntax

```
#include <ctools.h>
int recv
(
      int socketDescriptor,
      char * bufferPtr,
      int bufferLength,
      int flags
);
```

Function Description

recv is used to receive messages from another socket. **recv** may be used only on a *connected* socket (see **connect**, **accept**). *socketDescriptor* is a socket created with **socket** or **accept**. The length of the message is returned. If a message is too long to fit in the supplied buffer, excess bytes may be discarded depending on the type of socket the message is received from (see **socket**). The length of the message returned could also be smaller than *bufferLength* (this is not an error). If no messages are available at the socket, the receive call waits for a message to arrive, unless the socket is non-blocking, or the MSG_DONTWAIT flag is set in the *flags* parameter, in which case -1 is returned with socket error being set to EWOULDBLOCK.

Out-of-band data not in the stream (urgent data when the SO_OOBINLINE option is not set (default)) (TCP protocol only).

A single out-of-band data byte is provided with the TCP protocol when the SO_OOBINLINE option is not set. If an out-of-band data byte is present, recv with the MSG_OOB flag *not set* will not read past the position of the out-of-band data byte in a single **recv** request. That is, if there are 10 bytes from the current read position until the out-of-band byte, and if we execute a **recv** specifying a bufferLength of 20 bytes, and a flag value of 0, **recv** will only return 10 bytes. This forced stopping is to allow us to determine when we are at the out-of-band byte mark. When we are at the mark, **recv** with the MSG_OOB flag set can read the out-of-band data byte. Note that the user needs to use **select** in order to know when out-of-band data has arrived, or is arriving.

Out-of-band data (when the SO_OOBINLINE option is set (see setsockopt)).

(TCP protocol only)

If the SO_OOBINLINE option is enabled, the out-of-band data is left in the normal data stream and is read without specifying the MSG_OOB. More than one out-of-band data bytes can be in the stream at any given time. The out-of-band byte mark corresponds to the final byte of out-of-band data that was received. In this case, the MSG_OOB flag cannot be used with **recv**. The out-of- band data will be read in line with the other data. Again, **recv** will not read past the position of the out-of-band mark in a single **recv** request. Note that the user needs to use **select** in order to know when out-of-band data has arrived, or is arriving.

select may be used to determine when more data arrives, or/and when out-of-band data arrives.

Parameters

socketDescriptor	The socket descriptor to receive data from.
bufferPtr	The buffer to put the received data into

bufferLength	The length of the buffer area that *bufferPtr* points to
flags	See below

The *flags* parameter is formed by ORing one or more of the following:

MSG_DONTWAIT	Don't wait for data, but rather return immediately
MSG_OOB	Read any "out-of-band" data present on the socket rather than the regular "in-band" data
MSG_PEEK	"Peek" at the data present on the socket; the data is returned, but not consumed, so that a subsequent receive operation will see the same data.

Returns

>0	Number of bytes actually received from the socket.
0	EOF
-1	An error occurred

recv will fail if:

EBADF	The socket descriptor is invalid
ENOBUFS	There was insufficient user memory available to complete the operation
EMSGSIZE	The socket requires that message be received atomically, and bufferLength was too small
EWOULDBLOCK	The socket is marked as non-blocking or the MSG_DONTWAIT flag is used and no data is available to be read, or the MSG_OOB flag is set and the out of band data has not arrived yet from the peer
ESHUTDOWN	The remote socket has closed the connection, and there is no more data to be received (TCP socket only)
EINVAL	One of the parameters is invalid, or the MSG_OOB flag is set and, either the SO_OOBINLINE option is set, or there is no out of band data to read or coming from the peer
ENOTCONN	Socket is not connected.

recvfrom

Syntax

```
#include <ctools.h>
int recvfrom(
       int socketDescriptor,
       char * bufferPtr,
       int bufferLength,
       int flags,
       struct sockaddr * fromPtr,
       int * fromLengthPtr);
```

Function Description

recvfrom is used to receive messages from another socket. **recvfrom** may be used to receive data on a socket whether it is in a connected state or not but not on a TCP socket. *socketDescriptor* is a socket created with **socket**. If *fromPtr* is not a NULL pointer, the source address of the message is filled in. *fromLengthPtr* is a value-result parameter, initialized to the size of the buffer associated with *fromPtr*, and modified on return to indicate the actual size of the address stored there. The length of the message is returned. If a message is too long to fit in the supplied buffer, excess bytes may be discarded depending on the type of socket the message is received from (see **socket**). If no messages are available at the socket, the receive call waits for a message to arrive, unless the socket is non-blocking, or the MSG_DONTWAIT flag is set in the *flags* parameter, in which case -1 is returned with socket error being set to EWOULDBLOCK.

select may be used to determine when more data arrives, or/and when out-ofband data arrives.

Parameters

socketDescriptor	The socket descriptor to receive data from.
bufferPtr	The buffer to put the received data into
bufferLength	The length of the buffer area that *bufferPtr* points to
flags	See Below
fromPtr	The socket the data is (or to be) received from
fromLengthPtr	The length of the data area the *fromPtr* points to then upon return the actual length of the from data

The *flags* parameter is formed by ORing one or more of the following:

MSG_DONTWAIT	Don't wait for data, but rather return immediately
MSG_PEEK	"Peek" at the data present on the socket; the data is returned, but not consumed, so that a subsequent receive operation will see the same data.

Returns

>0	Number of bytes actually received from the socket.
0	EOF
-1	An error occurred

recvfrom will fail if:

EBADF	The socket descriptor is invalid.
EINVAL	One of the parameters is invalid.
EMSGSIZE	The socket requires that message be received atomically, and bufferLength was too small.
EPROTOTYPE	TCP protocol requires usage of **recv**, not **recvfrom**.
ENOBUFS	There was insufficient user memory available to comp lete the operation.
EWOULDBLOCK	The socket is marked as non-blocking and no data is available to be read.

release_processor

Release Processor to other Tasks

Syntax

```
#include <ctools.h>
void release_processor(void);
```

Description

The **release_processor** function releases control of the CPU to other tasks. Other tasks of the same priority will run. Tasks of the same priority run in a round-robin fashion using a time slicing mechanism. **release_processor** puts the task explicitely at the end of the round-robin-queue.

Notes

Calling **release_processor** in all idle loops is not necessary anymore. In contrary, it reduces the fair share of CPU time because the CPU is given up before the end of the time slice. The function **release_processor** still makes sense if the calling task does not have anything to do for the moment.

Release all resources in use by a task before releasing the processor.

Refer to the **Real Time Operating System** section for more information on tasks and task scheduling.

See Also

request_resource

release_resource

Release Control of a Resource

Syntax

```
#include <ctools.h>
void release_resource(UINT32 resource);
```

Description

The **release_resource** function releases control of the resource specified by *resource*.

If other tasks are waiting for the resource, the highest priority of these tasks, is given the resource and is made ready to execute. If no tasks are waiting the resource is made available, and the current task continues to run.

Notes

Refer to the **Real Time Operating System** section for more information on resources.

See Also

request_resource

Example

See the example for the **request_resource** function.

removeModbusHandler

Removes a User Defined Modbus Handler

Syntax

```
#include <ctools.h>
BOOLEAN removeModbusHandler(
UINT16 (* handler)(UCHAR *, UINT16,
                   UCHAR *, UINT16 *)
      );
```

Description

The **removeModbusHandler** function allows user-defined extensions to standard Modbus protocol to be removed. This function specifies the previously installed function that is to be removed.

This function returns TRUE if the specified handler was removed, and FALSE if the specified handler is not present.

Notes

This function is used to remove a user-defined extension to the standard Modbus protocol.

See Also

installModbusHandler

report_error

Set Task Error Code

Syntax

```
#include <ctools.h>
void report_error(UINT32 error);
```

Description

The **report_error** functions sets the error code for the current task to *error*. An error code is maintained for each executing task.

Notes

This function is used in sharable I/O routines to return error codes to the task using the routine.

Some functions supplied with the Microtec C compiler report errors using the global variable **errno**. The error code in this variable may be written over by another task before it can be used.

request_resource

Obtain Control of a Resource

Syntax

```
#include <ctools.h>
void request_resource(UINT32 resource);
```

Description

The **request_resource** function obtains control of the resource specified by *resource*. If the resource is in use, the task is blocked until it is available.

Notes

Use the **request_resource** function to control access to non-sharable resources. Refer to the **Real Time Operating System** section for more information on resources.

See Also

release_resource

Example

This code fragment obtains the dynamic memory resource, allocates some memory, and releases the resource.

```
#include <ctools.h>

void task(void)
{
       unsigned *ptr;

       /* ... code here */

       request_resource(DYNAMIC_MEMORY);
       ptr = (unsigned *)malloc((size_t)100);
       release_resource(DYNAMIC_MEMORY);

       /* ... more code here */
}
```

resetAllABSlaves

Erase All DF1 Slave Responses

Syntax

```
#include <ctools.h>
UINT16 resetAllABSlaves(FILE *stream);
```

Description

The **resetAllABSlaves** function is used to send a protocol message to all slaves communicating on the specified port to erase all responses not yet polled. *stream* specifies the serial port.

This function applies to the DF1 Half Duplex protocols only. The function returns **FALSE** if the protocol currently installed on the specified serial port is not a DF1 Half Duplex protocol, otherwise it returns **TRUE**.

Notes

The purpose of this command is to re-synch slaves with the master if the master has lost track of the order of responses to poll. This situation may exist if the master has been power cycled, for example.

See the example in the Example Programs chapter under the section **Master Message Example Using DF1 Protocol.** The **resetAllABSlaves** function should not normally be needed if polling is done using the sample polling function "poll_for_response" shown in this example.

See Also

pollABSlave

Example

This program segment will cause all slaves communicating on the **com2** serial port to erase all pending responses.

```
#include <ctools.h>

resetAllABSlaves(com2);
```

resetClockAlarm

Acknowledge and Reset Real Time Clock Alarm

Syntax

```
#include <ctools.h>
void resetClockAlarm(void);
```

Description

Real time clock alarms occur once after being set. The alarm setting remains in the real time clock. The alarm must be acknowledged before it can occur again.

The **resetClockAlarm** function acknowledges the last real time clock alarm and re-enables the alarm.

Notes

This function should be called after a real time clock alarm occurs.

The IO_SYSTEM resource must be requested before calling this function.

Example

See the example for the **installClockHandler** function.

route

Redirect Standard I/O Streams

Syntax

```
#include <ctools.h>
void route(UCHAR logical, UCHAR hardware);
```

Description

The **route** function redirects the I/O streams associated with `stdout`, `stdin`, and `stderr`. These streams are routed to the com1 serial port by default. *logical* specifies the stream to redirect. *hardware* specifies the hardware device which will output the data. It may be one of com1, com2, or com3.

Notes

This function has a global effect, so all tasks must agree on the routing.

Output streams must be redirected to a device that supports output. Input streams must be redirected to a device that supports input.

The use of this function is strongly discouraged since tasks beyond the control of the C Application may make use of the streams stdout, stdin, and stderr. This may result in data being unexpectedly added or removed from these streams.

Example

This program segment will redirect all input, output and errors to the **com2** serial port.

```
#include <ctools.h>

route(STD_ERR, com2);  /* send errors to com2 */
route(STD_OUT, com2);  /* send output to com2 */
route(STD_IN, com2);      /* get input from com2 */
```

rresvport

Syntax

```
#include <ctools.h>
int rresvport
(
      int * portToReservePtr
);
```

Function Description

rresvport is used to create a TCP socket and bind a reserved port to the socket starting with the port to reserve given by the user. The *portToReservePtr* parameter is a value result parameter. The integer pointed to by *portToReservePtr* is the first port number that the function attempts to bind to. The caller typically initializes the starting port number to IPPORT_RESERVED – 1. (IPPORT_RESERVED is defined as 1024.) If the bind fails because that port is already used, then **rresvport** decrements the port number and tries again. If it finally reaches IPPORT_RESERVEDSTART (defined as 600) and finds it already in use, it returns –1 and set the socket error to EAGAIN. If this function successfully binds to a reserved port number, it returns the socket descriptor to the user and stores the reserved port that the socket is bound to in the integer cell pointed to by *portToReservePtr.*

Parameters

portToReservePtr	Pointer to the port number to reserve, and to the port number reserved on success.

Returns

>= 0	Valid socket descriptor
-1	An error occurred.

If an error occurred, the socket error can be retrieved by calling **getErrorCode**(socketDescriptor).

rresvport will fail if:

EAGAIN	The TCP/IP stack could not find any port number available between IPPORT_RESERVEDSTART and the port number to reserve.
EINVAL	Bad parameter; pointer is null or port number to reserve is less than IPPORT_RESERVEDSTART (600).

runBackgroundIO

Run Background I/O Task

Syntax

```
#include <ctools.h>
void runBackgroundIO( BOOLEAN state );
```

Description

The **runBackgroundIO** function is used to start or stop the Background I/O task. This task provides dialup support and controls the LED Power pushbutton.

Calling the function with the argument *state* set to FALSE stops the Background I/O task. Calling the function with *state* set to TRUE starts the task.

This function should only be needed in the context of the startup function **appstart**.

runIOSystem

Run I/O System

Syntax

```
#include <ctools.h>
void runIOSystem( BOOLEAN state );
```

Description

The **runIOSystem** function is used to start or stop the I/O System tasks. The I/O System must be running to access I/O modules through the functions in the `ioRead` and `ioWrite` group.

Calling the function with the argument *state* set to FALSE stops the I/O System. Calling the function with *state* set to TRUE starts the I/O System.

This function should only be needed in the context of the startup function **appstart**.

runLed

Control Run LED State

Syntax

```
#include <ctools.h>
void runLed(UINT16 state);
```

Description

The **runLed** function sets the run light LED to the specified state. *state* may be one of the following values.

LED_ON turn on run LED
LED_OFF turn off run LED

The run LED remains in the specified state until changed, or until the controller is reset.

Notes

The ladder logic interpreter controls the state of the RUN LED. If ladder logic is installed in the controller, this function should not be used by a C program.

Example

```
#include <ctools.h>

int main(void)
{
      runLed(LED_ON);     /* program is running */
      /* ... the rest of the code */
}
```

runMasterIpStartTask

Run TCP/IP Master Message Support Task

Syntax

```
#include <ctools.h>
void runMasterIpStartTask( BOOLEAN state );
```

Description

The **runMasterIpStartTask** function is used to start or stop the TCP/IP master message support task. This task must be running to allow master messaging over a TCP/IP network using the functions in the `mTcpMaster` group.

Calling the function with the argument *state* set to FALSE stops the task. Calling the function with *state* set to TRUE starts the task.

This function should only be needed in the context of the startup function **appstart**.

See Also

mTcpMasterMessage

runTarget

Start the Run-Time Engine

Syntax

```
#include <ctools.h>
void runTarget( BOOLEAN state );
```

Description

The **runTarget** function is used to start or stop the run-time engine task. For TelePACE firmware, this is the Ladder Logic run-time engine. For ISaGRAF firmware this is the ISaGRAF IEC 1131 run-time engine.

Calling the function with the argument *state* set to FALSE stops the run-time engine task. Calling the function with *state* set to TRUE starts the task.

This function should only be needed in the context of the startup function **appstart**.

select

Syntax

```
#include <ctools.h>
int select
(
      int numberSockets,
      fd_set * readSocketsPtr,
      fd_set * writeSocketsPtr,
      fd_set * exceptionSocketsPtr,
      struct timeval * timeOutPtr
);
```

Function Description

select examines the socket descriptor sets whose addresses are passed in *readSocketsPt*r, *writeSocketsPt*r, and *exceptionSocketsPtr* to see if any of their socket descriptors are ready for reading, are ready for writing, or have an exceptional condition pending, respectively. Out-of-band data is the only exceptional condition. The *numberSockets* argument specifies the number of socket descriptors to be tested. Its value is the maximum socket descriptor to be tested, plus one. The socket descriptors from 0 to *numberSockets* -1 in the socket descriptor sets are examined. On return, select replaces the given socket descriptor sets with subsets consisting of those socket descriptors that are ready for the requested operation. The return value from the call to select is the number of ready socket descriptors. The socket descriptor sets are stored as bit fields in arrays of integers.

The following macros are provided for manipulating such file descriptor sets:

FD_ZERO(&fdset);	Initializes a socket descriptor set (*fdset)* to the null set.
FD_SET(fd, &fdset);	Includes a particular socket descriptor *fd* in *fdse*t.
FD_CLR(fd, &fdset);	Removes *fd* from *fdse*t.
FD_ISSET(fd, &fdset);	Is non-zero if *fd* is a member of *fdse*t, zero otherwise.

Note the term “fd” is used for BSD compatibility since select is used on both file systems and sockets under BSD Unix.

Parameters

numberSockets	Biggest socket descriptor to be tested, plus one.
readSocketsPtr	The pointer to a mask of sockets to check for a read condition.
writeSocketsPtr	The pointer to a mask of sockets to check for a write condition.
exceptionSocketsPtr	The pointer to a mask of sockets to check for an exception condition: Out of Band data.
timeOutPtr	The pointer to a structure containing the length of time to wait for an event before exiting.

Returns

>0	Number of sockets that are ready
0	Time limit exceeded
-1	An error occurred

If an error occurred, the socket error can be retrieved by calling **getErrorCode**(socketDescriptor).

select will fail if:

EBADF	One of the socket descriptors is bad.
EINVAL	A component of the pointed-to time limit is outside the acceptable range: tv_sec must be between 0 and 10^8, inclusive. tv_usec must be greater than or equal to 0, and less than 10^6.

send

Syntax

```
#include <ctools.h>
int send
(
       int socketDescriptor,
       char * bufferPtr,
       int bufferLength,
       int flags
);
```

Function Description

send is used to transmit a message to another transport end-point. **send** may be used only when the socket is in a *connected* state. *socketDescriptor* is a socket created with **socket**.

If the message is too long to pass atomically through the underlying protocol (non TCP protocol), then the error EMSGSIZE is returned and the message is not transmitted.

A return value of -1 indicates locally detected errors only. A positive return value does not implicitly mean the message was delivered, but rather that it was sent.

Blocking socket send: if the socket does not have enough buffer space available to hold the message being sent, **send** blocks.

Non blocking stream (TCP) socket send: if the socket does not have enough buffer space available to hold the message being sent, the send call does not block. It can send as much data from the message as can fit in the TCP buffer and returns the length of the data sent. If none of the message data fits, then –1 is returned with socket error being set to EWOULDBLOCK.

Non blocking datagram socket send: if the socket does not have enough buffer space available to hold the message being sent, no data is being sent and -1 is returned with socket error being set to EWOULDBLOCK.

The **select** call may be used to determine when it is possible to send more data.

Sending Out-of-Band Data:

For example, if you have remote login application, and you want to interrupt with a ^C keystroke, at the socket level you want to be able to send the ^C flagged as special data (also called out-of-band data). You also want the TCP protocol to let the peer (or remote) TCP know as soon as possible that a special character is coming, and you want the peer (or remote) TCP to notify the peer (or remote) application as soon as possible. At the TCP level, this mechanism is called TCP urgent data. At the socket level, the mechanism is called out-of-band data. Out-of-band data generated by the socket layer, is implemented at the TCP layer with the urgent data mechanism. The user application can send one or several out-of-band data bytes. With TCP you cannot send the out-of-band data ahead of the data that has already been buffered in the TCP send buffer, but you can let the other side know (with the urgent flag, i.e the term urgent data) that out-of-band data is coming, and you can let the peer TCP know the offset of the current data to the last byte of out-of-band data. So with TCP, the out-of-band data byte(s) are not sent ahead of the data stream, but the TCP protocol can notify the remote TCP ahead of time that some out-of-band data byte(s) exist. What TCP does, is mark the byte stream where urgent data ends, and set the Urgent flag bit in the TCP header flag field, as long as it is sending data before ,or up to, the last byte of out-of-band data.

In your application, you can send out-of-band data, by calling the **send** function with the MSG_OOB flag. All the bytes of data sent that way (using **send** with the MSG_OOB flag) are out-of-band data bytes. Note that if you call **send** several times with out-of-band data, TCP will always keep track of where the last out-of-band byte of data is in the byte data stream, and flag this byte as the last byte of urgent data. To receive out-of-band data, please see the recv section of this manual.

Parameters

socketDescriptor	The socket descriptor to use to send data
bufferPtr	The buffer to send
bufferLength	The length of the buffer to send
flags	See below

The *flags* parameter is formed by ORing one or more of the following:

MSG_DONTWAIT	Don't wait for data send to complete, but rather return immediately
MSG_OOB	Send "out-of-band" data on sockets that support this notion. The underlying protocol must also support "out-of-band" data. Only SOCK_STREAM sockets created in the AF_INET address family support out-of-band data
MSG_DONTROUTE	The SO_DONTROUTE option is turned on for the duration of the operation. Only diagnostic or routing programs use it

Returns

>=0	Number of bytes actually sent on the socket
-1	An error occurred.

send will fail if:

EBADF	The socket descriptor is invalid.
ENOBUFS	There was insufficient user memory available to complete the operation.
EHOSTUNREACH	Non-TCP socket only. No route to destination host.
EMSGSIZE	The socket requires that message to be sent atomically, and the message was too long.
EWOULDBLOCK	The socket is marked as non-blocking and the send operation would block.
ENOTCONN	Socket is not connected.
ESHUTDOWN	User has issued a write **shutdown** (TCP socket only).

send_message

Send a Message to a Task

Syntax

```
#include <ctools.h>
void send_message(envelope *penv);
```

Description

The **send_message** function sends a message to a task. The envelope specified by *penv* contains the message destination, type and data.

The envelope is placed in the destination task's message queue. If the destination task is waiting for a message it is made ready to execute.

The current task is not blocked by the **send_message** function.

Notes

Envelopes are obtained from the operating system with the **allocate_envelope** function.

See Also

receive_message

Example

This program creates a task to display a message and sends a message to it.

```
#include <ctools.h>

void showIt(void)
{
       envelope *msg;

       while (TRUE)
       {
          msg = receive_message();
          fprintf(com1,"Message data %ld\r\n", msg->data);
          deallocate_envelope(msg);
       }
}

int main(void)
{
       envelope *msg;      /* message pointer */
       unsigned tid;       /* task ID */

       tid = create_task(showIt, 100, APPLICATION, 4);
       msg = allocate_envelope();
       msg->destination = tid;
       msg->type        = MSG_DATA;
       msg->data        = 1002;
       send_message(msg);

       /* wait for ever so that main and other
       APPLICATION tasks won't end */
       while(TRUE)
```

```
        {
            /* Allow other tasks to execute */
            release_processor();
        }
}
```

sendto

Syntax

```
#include <trsocket.h>
int sendto
(
      int socketDescriptor,
      char * bufferPtr,
      int bufferLength,
      int flags,
      const struct sockaddr * toPtr,
      int toLength
);
```

Function Description

sendto is used to transmit a message to another transport end-point. **sendto** may be used at any time (either in a connected or unconnected state), but not for a TCP socket. *socketDescriptor* is a socket created with **socket**. The address of the target is given by *to* with *toLength* specifying its size. If the message is too long to pass atomically through the underlying protocol, then –1 is returned with the socket error being set to EMSGSIZE, and the message is not transmitted.

A return value of -1 indicates locally detected errors only. A positive return value does not implicitly mean the message was delivered, but rather that it was sent.

If the socket does not have enough buffer space available to hold the message being sent, and is in blocking mode, **sendto** blocks. If it is in non-blocking mode or the MSG_DONTWAIT flag has been set in the *flags* parameter, –1 is returned with the socket error being set to EWOULDBLOCK. The **select** call may be used to determine when it is possible to send more data.

Parameters

socketDescriptor The socket descriptor to use to send data.

bufferPtr	The buffer to send.
bufferLength	The length of the buffer to send.
toPtr	The address to send the data to.
toLength	The length of the to area pointed *to by toPtr.*
flags	See below

The *flags* parameter is formed by ORing one or more of the following:

MSG_DONTWAIT	Don't wait for data send to complete, but rather return immediately.
MSG_DONTROUTE	The SO_DONTROUTE option is turned on for the duration of the operation. Only diagnostic or routing programs use it.

Returns

Value	Meaning
>=0	Number of bytes actually sent on the socket
-1	An error occurred

sendto will fail if:

EBADF	The socket descriptor is invalid. ENOBUFS There was insufficient user memory available to complete the operation. EHOSTUNREACH No route to destination host. EMSGSIZE The socket requires that message be sent atomically, and the message was too long.
EPROTOTYPE	TCP protocol requires usage of **send** not **sendto**.
EWOULDBLOCK	The socket is marked as non-blocking and the send operation would block.

serialModbusMaster

Send Modbus Command

Syntax

```
#include <ctools.h>
       BOOLEAN serialModbusMaster( MASTER_MESSAGE * message,
       MODBUS_SESSION * session );
```

Description

The **serialModbusMaster** function sends a command on a serial port using a Modbus protocol. The Modbus protocol task waits for the response from the slave station. The current task continues execution.

- *message* points to a MASTER_MESSAGE structure defining the message parameters and serial port to use. MASTER_MESSAGE is described in the *Structures and Types* section.
- *session* points to a MODBUS_SESSION structure. This structure is used by the Modbus protocol task. Declare the MODBUS_SESSION structure as a static modular or global variable. A local variable or dynamically allocated variable may not be used because a late command response received after the variable is freed will write data over the freed variable space.

The **serialModbusMaster** function returns TRUE if a valid message has been queued for transmission. The function returns FALSE if the message definition is invalid or the transmission queue is full. Refer to the *session->masterCmdStatus* field for an error code. Error codes are described in the *Structures and Types* section for the enum MODBUS_CMD_STATUS.

The calling task monitors the status of the command sent using the *session->masterCmdStatus* field. The *masterCmdStatus* field is set to **MM_SENT** if a master message is sent. It will be set to **MM_RECEIVED** when the response to the message is received.

The command status will be set to **MM_RSP_TIMEOUT** if the response is not received within the specified timeout. The application must wait for a status of **MM_RECEIVED** or **MM_RSP_TIMEOUT** before sending the next master message.

This function may be used at the same time on the same serial port as a TelePACE MSTR element or ISaGRAF master function block.

Notes

Refer to the communication protocol manual for more information.

To optimize performance, minimize the length of messages on com3. Examples of recommended uses for com3 are for local operator display terminals, and for programming and diagnostics using the ISaGRAF program.

The IO_SYSTEM resource must be requested before calling this function.

See Also

get_protocol_status, clear_protocol_status, master_message

Example

See the example in the Example Programs chapter under the section **Master Message Example Using serialModbusMaster.**

setABConfiguration

Set DF1 Protocol Configuration

Syntax

```
#include <ctools.h>
UINT16 setABConfiguration(FILE *stream, struct    ABConfiguration
*ABConfig);
```

Description

The **setABConfiguration** function sets DF1 protocol configuration parameters. *stream* specifies the serial port. *ABConfig* references a DF1protocol configuration structure. Refer to the description of the ABConfiguration structure for an explanation of the fields.

The **setABConfiguration** function returns **TRUE** if the settings were changed. It returns **FALSE** if *stream* does not point to a valid serial port.

See Also

getABConfiguration

Example

This code fragment changes the maximum protected address to 7000. This is the maximum address accessible by protected DF1 commands received on com2.

```
#include <ctools.h>
struct ABConfiguration ABConfig;

getABConfiguration(com2, &ABConfig);
ABConfig.max_protected_address = 7000;
setABConfiguration(com2, &ABConfig);
```

setclock

Set Real Time Clock

Syntax

```
#include <ctools.h>
void setclock(TIME *now);
```

Description

The **setclock** function sets the real time clock. *now* references a TIME structure containing the time and date to be set.

Refer to the **Structures and Types** section for a description of the fields. All fields of the clock structure must be set with valid values for the clock to operate properly.

Notes

The IO_SYSTEM resource must be requested before calling this function.

See Also

getclock

Example

This function switches the clock to daylight savings time.

```
#include <ctools.h>

void daylight(void)
{
       TIME now;

       request_resource(IO_SYSTEM);
       getclock(&now);
       now.hour = now.hour + 1 % 24;
       setclock(&now);
       request_resource(IO_SYSTEM);
}
```

setClockAlarm

Set the Real Time Clock Alarm

Syntax

```
#include <ctools.h>
UINT16 setClockAlarm(ALARM_SETTING alarm);
```

Description

The **setClockAlarm** function configures the real time clock to alarm at the specified alarm setting. The ALARM_SETTING structure *alarm* specifies the time of the alarm. Refer to the *ctools.h* section for a description of the fields in the structure.

The function returns TRUE if the alarm can be configured, and FALSE if there is an error in the alarm setting. No change is made to the alarm settings if there is an error.

Notes

An alarm will occur only once, but remains set until disabled. Use the **resetClockAlarm** function to acknowledge an alarm that has occurred and re-enable the alarm for the same time.

Set the alarm type to AT_NONE to disable an alarm. It is not necessary to specify the hour, minute and second when disabling the alarm.

The IO_SYSTEM resource must be requested before calling this function.

See Also

getClockAlarm

setdbase

Write Value to I/O Database

Syntax

```
#include <ctools.h>
void setdbase(UINT16 type, UINT16 address, INT16 value);
```

Description

The **setdbase** function writes *value* to the I/O database. *type* specifies the method of addressing the database. *address* specifies the location in the database. The table below shows the valid address types and ranges

Type	Address Ranges	Register Size
MODBUS	00001 to NUMCOIL 10001 to 10000 + NUMSTATUS 30001 to 30000 + NUMINPUT 40001 to 40000 + NUMHOLDING	1 bit 1 bit 16 bit 16 bit
LINEAR	0 to NUMLINEAR-1	16 bit

Notes

If the specified register is currently forced, the I/O database remains unchanged.

When writing to LINEAR digital addresses, *value* is a bit mask which writes data to 16 1-bit registers at once. If any of these 1-bit registers is currently forced, only the forced registers remain unchanged.

The I/O database is not modified when the controller is reset. It is a permanent storage area, which is maintained during power outages.

Refer to the **Functions Overview** section for more information.

The IO_SYSTEM resource must be requested before calling this function.

Example

```
#include <ctools.h>

int main(void)
{
       request_resource(IO_SYSTEM);

       setdbase(MODBUS, 40001, 102);

       /* Turn ON the first 16 coils */
       setdbase(LINEAR, START_COIL, 255);

       /* Write to a 16 bit register */
       setdbase(LINEAR, 3020, 240);

       /* Write to the 12th holding register */
       setdbase(LINEAR, START_HOLDING, 330);

       /* Write to the 12th holding register */
       setdbase(LINEAR, START_HOLDING, 330);
```

```
        release_resource(IO_SYSTEM);
}
```

Setdbase Handler Function

User Defined Setdbase Handler Function

The setdbase handler function is a user-defined function that handles writing to Modbus addresses not assigned in the ISaGRAF Dictionary. The function can have any name; *setdbaseHandler* is used in the description below.

Syntax

```
#include <ctools.h>
BOOLEAN setdbaseHandler(
   UINT16 address,
   INT16 value
   )
```

Description

This function is called by the **setdbase** function when one of the following conditions apply:

- There is no ISaGRAF application downloaded, or
- There is no ISaGRAF variable assigned to the specified Modbus address.

The function has two parameters:

- The *address* parameter is the Modbus address to be written.
- The *value* parameter is the integer value to write to the Modbus *address*.

If the address is to be handled, the handler function must return TRUE and write *value* to the current value at the Modbus *address*.

If the address is not to be handled, the function must return FALSE and do nothing.

Notes

The IO_SYSTEM resource must be requested before calling setdbase, which calls this handler. Requesting the IO_SYSTEM resource ensures that only one task may call the handler at a time. Therefore, the function does not have to be re-entrant.

An array may be defined to store the current values for all Modbus addresses handled by this function. See the section *Data Storage* if a non-initialized data array is required.

See Also

installSetdbaseHandler

setDTR

Control RS232 Port DTR Signal

Syntax

```
#include <ctools.h>
void setDTR(UCHAR port, UINT16 state);
```

Description

The **setDTR** function sets the status of the DTR signal line for the communication port specified by *port*. When *state* is SIGNAL_ON the DTR line is asserted. When *state* is SIGNAL_OFF the DTR line is de-asserted.

Notes

The DTR line follows the normal RS232 voltage levels for asserted and de-asserted states.

This function is only useful on RS232 ports. The function has no effect if the serial port is not an RS232 port.

setForceFlag

Set Force Flag State for a Register (TelePACE firmware only)

Syntax

```
#include <ctools.h>
BOOLEAN setForceFlag(UINT16 type, UINT16 address, UINT16 value);
```

Description

The **setForceFlag** function sets the force flag(s) for the specified database register(s) to *value*. *value* is either 1 or 0, or a 16-bit mask for LINEAR digital addresses. The valid range for *address* is determined by the database addressing *type*.

If the *address* or addressing *type* is not valid, force flags are left unchanged and FALSE is returned; otherwise TRUE is returned. The table below shows the valid address types and ranges.

Type	Address Ranges	Register Size
MODBUS	00001 to NUMCOIL 10001 to 10000 + NUMSTATUS 30001 to 30000 + NUMINPUT 40001 to 40000 + NUMHOLDING	1 bit 1 bit 16 bit 16 bit
LINEAR	0 to NUMLINEAR-1	16 bit

Notes

When a register's force flag is set, the value of the I/O database at that register is forced to its current value. This register's value can only be modified by using the **overrideDbase** function or the *Edit/Force Register dialog.* While forced this value cannot be modified by the **setdbase** function, protocols, or Ladder Logic programs.

Force Flags are not modified when the controller is reset. Force Flags are in a permanent storage area, which is maintained during power outages.

The IO_SYSTEM resource must be requested before calling this function.

See Also

clearRegAssignment

getForceFlag

getOutputsInStopMode

overrideDbase

Example

This program clears the force flag for register 40001 and sets the force flags for the 16 registers at linear address 302 (i.e. registers 10737 to 10752).

```
#include <ctools.h>

int main(void)
{
      request_resource(IO_SYSTEM);
```

```
        setForceFlag(MODBUS, 40001, 0);
        setForceFlag(LINEAR, 302, 255);

        release_resource(IO_SYSTEM);
}
```

setIOErrorIndication

Set I/O Module Error Indication

Syntax

```
#include <ctools.h>
void setIOErrorIndication(BOOLEAN state);
```

Description

The **setIOErrorIndication** function sets the I/O module error indication to the specified *state*. If set to TRUE, the I/O module communication status is reported in the controller status register and Status LED. If set to FALSE, the I/O module communication status is not reported.

To save these settings with the controller settings in flash memory so that they are loaded on controller reset, call **flashSettingsSave** as shown below.

```
request_resource(FLASH_MEMORY);
flashSettingsSave(CS_PERMANENT);
release_resource(FLASH_MEMORY);
```

Notes

Refer to the ***5203/4 System Manual***, ***SCADAPack 32 System Manual,*** or ***SCADAPack2 System Manual*** for further information on the Status LED and Status Output.

See Also

getIOErrorIndication

setOutputsInStopMode

Set Outputs In Stop Mode (TelePACE firmware only)

Syntax

```
#include <ctools.h>
void setOutputsInStopMode(
   BOOLEAN holdDoutsOnStop,
   BOOLEAN holdAoutsOnStop
   );
```

Description

The **setOutputsInStopMode** function sets the *holdDoutsOnStop* and *holdAoutsOnStop* control flags to the specified state.

If *holdDoutsOnStop* is set to TRUE, then digital outputs are held at their last state when the Ladder Logic program is stopped. If *holdDoutsOnStop* is FALSE, then digital outputs are turned OFF when the Ladder Logic program is stopped.

If *holdAoutsOnStop* is TRUE, then analog outputs are held at their last value when the Ladder Logic program is stopped. If *holdAoutsOnStop* is FALSE, then analog outputs go to zero when the Ladder Logic program is stopped.

To save these settings with the controller settings in flash memory so that they are loaded on controller reset, call **flashSettingsSave** as shown below.

```
request_resource(FLASH_MEMORY);
flashSettingsSave(CS_PERMANENT);
release_resource(FLASH_MEMORY);
```

See Also

getOutputsInStopMode

Example

This program changes the output conditions to hold analog outputs at their last value when the Ladder Logic program is stopped.

```
#include <ctools.h>

int main(void)
{
       unsigned holdDoutsOnStop;
       unsigned holdAoutsOnStop;
       getOutputsInStopMode( &holdDoutsOnStop, &holdAoutsOnStop);
       holdAoutsOnStop = TRUE;
       setOutputsInStopMode( holdDoutsOnStop,  holdAoutsOnStop);
}
```

set_port

Set Serial Port Configuration

Syntax

```
#include <ctools.h>
void set_port(UCHAR port, struct pconfig *settings);
```

Description

The **set_port** function sets serial port communication parameters. *port* must specify one of **com1**, **com2**, or **com3**. *settings* references a serial port configuration structure. Refer to the description of the `pconfig` structure for an explanation of the fields.

Notes

If the serial port settings are the same as the current settings, this function has no effect.

The serial port is reset when settings are changed. All data in the receive and transmit buffers are discarded.

The IO_SYSTEM resource must be requested before calling this function.

To save these settings with the controller settings in flash memory so that they are loaded on controller reset, call **flashSettingsSave** as shown below.

```
request_resource(FLASH_MEMORY);
flashSettingsSave(CS_RUN);
release_resource(FLASH_MEMORY);
```

See Also

get_port

Example

This code fragment changes the baud rate on com2 to 19200 baud.

```
#include <ctools.h>
struct pconfig settings;

get_port(com2, &settings);
settings.baud = BAUD19200;
request_resource(IO_SYSTEM);
set_port(com2, &settings);
release_resource(IO_SYSTEM);
```

This code fragment sets com2 to the same settings as com1.

```
#include <ctools.h>
struct pconfig settings;

request_resource(IO_SYSTEM);
set_port(com2, get_port(com1, &settings));
release_resource(IO_SYSTEM);
```

setPowerMode

Set Current Power Mode

Syntax

```
#include <ctools.h>
BOOLEAN setPowerMode(UCHAR cpuPower, UCHAR lan, UCHAR usbPeripheral,
UCHAR usbHost);
```

Description

The **setPowerMode** function returns TRUE if the new settings were successfully applied. The setPowerMode function allows for power savings to be realised by controlling the power to the LAN port, changing the clock speed, and individually controlling the host and peripheral USB power. The following table of macros summarizes the choices available.

Macro	***Meaning***
PM_CPU_FULL	The CPU is set to run at full speed
PM_CPU_REDUCED	The CPU is set to run at a reduced speed
PM_CPU_SLEEP	The CPU is set to sleep mode
PM_LAN_ENABLED	The LAN is enabled
PM_LAN_DISABLED	The LAN is disabled
PM_USB_PERIPHERAL_ENABLED	The USB peripheral port is enabled
PM_USB_PERIPHERAL_DISABLED	The USB peripheral port is disabled
PM_USB_HOST_ENABLED	The USB host port is enabled
PM_USB_HOST_DISABLED	The USB host port is disabled
PM_NO_CHANGE	The current value will be used

TRUE is returned if the requested change was made, otherwise FALSE is returned.

The application program may view the current power mode with the **getPowerMode** function.

See Also

getPowerMode, setWakeSource, getWakeSource

setProgramStatus

Set Program Status Flag

Syntax

```
#include <ctools.h>
void setProgramStatus(FUNCPTR entryPoint, UINT16 status);
```

Description

The **setProgramStatus** function sets the application program status flag. The status flag is set to **NEW_PROGRAM** when a cold boot of the controller is performed, or a program is downloaded to the controller from the program loader. The parameter entryPoint should always be set to the function `main`.

Notes

There are three pre-defined values for the flag. However the application program may make whatever use of the flag it sees fit.

NEW_PROGRAM	indicates the program is newly loaded.
PROGRAM_EXECUTED	indicates the program has been executed.
PROGRAM_NOT_LOADED	indicates that the requested program is not loaded

See Also

getProgramStatus

Example

See Get Program Status Example.

set_protocol

Set Communication Protocol Configuration

Syntax

```
#include <ctools.h>
INT16 set_protocol(UCHAR port, struct prot_settings *settings);
```

Description

The **set_protocol** function sets protocol parameters. *port* must specify one of **com1**, **com2** or **com3**. *settings* references a protocol configuration structure. Refer to the description of the `prot_settings` structure for an explanation of the fields.

The **set_protocol** function returns **TRUE** if the settings were changed. It returns **FALSE** if there is an error in the settings or if the protocol fails to start.

The IO_SYSTEM resource must be requested before calling this function.

To save these settings with the controller settings in flash memory so that they are loaded on controller reset, call **flashSettingsSave** as shown below.

```
request_resource(FLASH_MEMORY);
flashSettingsSave(CS_RUN);
release_resource(FLASH_MEMORY);
```

Notes

Setting the protocol type to NO_PROTOCOL ends the protocol task and frees the stack resources allocated to it.

Be sure to add a call to `modemNotification` when writing a custom protocol.

See Also

get_protocol

Example

This code fragment changes the station number of the com2 protocol to 4.

```
#include <ctools.h>
struct prot_settings settings;

get_protocol(com2, &settings);
settings.station = 4;
request_resource(IO_SYSTEM);
set_protocol(com2, &settings);
release_resource(IO_SYSTEM);
```

setProtocolSettings

Set Protocol Extended Addressing Configuration

Syntax

```
#include <ctools.h>
BOOLEAN setProtocolSettings(
      UCHAR port,
      PROTOCOL_SETTINGS * settings
      );
```

Description

The setProtocolSettings function sets protocol parameters for a serial port. This function supports extended addressing.

The function has two arguments: *port* is one of com1, com2, or com3; and *settings,* a pointer to a PROTOCOL_SETTINGS structure. Refer to the description of the structure for an explanation of the parameters.

The function returns **TRUE** if the settings were changed. It returns **FALSE** if the stream is not valid, or if the protocol fails to start.

The IO_SYSTEM resource must be requested before calling this function.

To save these settings with the controller settings in flash memory so that they are loaded on controller reset, call **flashSettingsSave** as shown below.

```
request_resource(FLASH_MEMORY);
flashSettingsSave(CS_RUN);
release_resource(FLASH_MEMORY);
```

Notes

Setting the protocol type to NO_PROTOCOL ends the protocol task and frees the stack resources allocated to it.

Be sure to add a call to `modemNotification` when writing a custom protocol.

Extended addressing is available on the Modbus RTU and Modbus ASCII protocols only. See the *TeleBUS Protocols User Manual* for details.

Example

This code fragment sets protocol parameters for the com2 serial port.

```
#include <ctools.h>
PROTOCOL_SETTINGS settings;

settings.type        = MODBUS_RTU;
settings.station     = 1234;
settings.priority    = 250;
settings.SFMessaging = FALSE;
settings.mode        = AM_extended;

request_resource(IO_SYSTEM);
setProtocolSettings(com2, &settings);
release_resource(IO_SYSTEM);
```

setProtocolSettingsEx

Sets extended protocol settings for a serial port.

Syntax

```
#include <ctools.h>
BOOLEAN setProtocolSettingsEx(
      UCHAR port,
      PROTOCOL_SETTINGS_EX * pSettings
      );
```

Description

The `setProtocolSettingsEx` function sets protocol parameters for a serial port. This function supports extended addressing and Enron Modbus parameters.

The function has two arguments:

- `port` specifies the serial port. It is one of com1, com2 or com3.
- `pSettings` is a pointer to a `PROTOCOL_SETTINGS_EX` structure. Refer to the description of the structure for an explanation of the parameters.

The function returns `TRUE` if the settings were changed. It returns `FALSE` if the stream is not valid, or if the protocol fails to start.

To save these settings with the controller settings in flash memory so that they are loaded on controller reset, call **flashSettingsSave** as shown below.

```
request_resource(FLASH_MEMORY);
flashSettingsSave(CS_RUN);
release_resource(FLASH_MEMORY);
```

Notes

The IO_SYSTEM resource must be requested before calling this function.

Setting the protocol type to NO_PROTOCOL ends the protocol task and frees the stack resources allocated to it.

Be sure to add a call to `modemNotification` when writing a custom protocol.

Extended addressing and the Enron Modbus station are available on the Modbus RTU and Modbus ASCII protocols only. See the *TeleBUS Protocols User Manual* for details.

Example

This code fragment sets protocol parameters for the com2 serial port.

```
#include <ctools.h>
PROTOCOL_SETTINGS_EX settings;

settings.type =         MODBUS_RTU;
settings.station =      1;
settings.priority =     250;
settings.SFMessaging =  FALSE;
settings.mode =         AM_standard;
settings.enronEnabled = TRUE;
```

```
settings.enronStation = 4;

request_resource(IO_SYSTEM);
setProtocolSettingsEx(com2, &settings);
release_resource(IO_SYSTEM);
```

setSFTranslation

Write Store and Forward Translation

Syntax

```
#include <ctools.h>
struct SFTranslationStatus setSFTranslation(UINT16 index, SF_TRANSLATION
    * pTranslation);
```

Description

Instead of using the **setSFTranslation** function use the **setSFTranslationEx** function, which supports translations with a timeout and with authentication. Otherwise a default timeout of 10 seconds is set for all forwarded commands.

The **setSFTranslation** function copies the structure pointed to by *pTranslation* into the store and forward translation table at the location specified by *index*. Valid values for *index* are 0 to 127. The function checks for invalid translations. The translation is always stored even if invalid.

The SF_TRANSLATION structure is described in the *Structures and Types* section.

The function returns a SFTranslationStatus structure. It is described in the *Structures and Types* section. The *code* field of the structure is set to one of the following. If there is an error, the *index* field is set to the location of the translation that is not valid.

Result code	***Meaning***
SF_VALID	All translations are valid
SF_NO_TRANSLATION	The entry defines re-transmission of the same message on the same port
SF_PORT_OUT_OF_RANGE	One or both of the serial port indexes is not valid
SF_STATION_OUT_OF_RANGE	One or both of the stations is not valid
SF_ALREADY_DEFINED	The translation already exists in the table
SF_INDEX_OUT_OF_RANGE	The entry referenced by index does not exist in the table
SF_INVALID_FORWARDING_IP	The forwarding IP address is invalid.

Notes

The ***TeleBUS Protocols User Manual*** describes the store and forward messaging mode.

Writing a translation with both stations set to station 65535 can clear a translation in the table. Station 65535 is not a valid station.

The Modbus protocol type and communication parameters may differ between serial ports. The store and forward messaging will translate the protocol messages.

Translations describe the communication path of the master command: e.g. the slave interface which receives the command and the forwarding interface to forward the command. The response to the command is automatically returned to master through the same communication path in reverse.

Do not specify additional entries in the Store and Forward Table to describe the response path.

The IO_SYSTEM resource must be requested before calling this function.

To save the Store and Forward Table with the controller settings in flash memory so that it is loaded on controller reset, call **flashSettingsSave** as shown below.

```
// save Store & Forward table with controller settings
request_resource(FLASH_MEMORY);
flashSettingsSave(CS_RUN);
release_resource(FLASH_MEMORY);
```

Translations may involve any combination of interfaces. The interfaces may be running a Serial Modbus or Modbus IP protocol.

Slave Interface	Forwarding Interface
Serial Modbus Interface: e.g. com1, com2, or com3	Serial Modbus Interface: e.g. com1, com2, or com3
Modbus IP Interface: e.g. Ethernet1	Serial Modbus Interface: e.g. com1, com2, or com3
Serial Modbus Interface: e.g. com1, com2, or com3	Modbus IP Network: e.g. Modbus/TCP, Modbus RTU over UDP, or Modbus ASCII over UDP
Modbus IP Interface: e.g. Ethernet1	Modbus IP Network: e.g. Modbus/TCP, Modbus RTU over UDP, or Modbus ASCII over UDP

Modbus IP Network as Forwarding Interface

When forwarding to a TCP or UDP network, the protocol type is selected for the *forwardInterface* in the SF_TRANSLATION structure. The IP Stack automatically determines the exact interface (e.g. Ethernet1) to use when it searches the network for the *forwardIPAddress*.

Also, when forwarding on a TCP or UDP network, the forwarding destination IP address must be entered as the *forwardIPAddress*. The *forwardIPAddress* is entered as an IP address string of the format 255.255.255.255. The *forwardIPAddress* is needed to know where to connect so that the command can be forwarded to its final destination.

Modbus IP Network as Slave Interface

Note that there is no field for a slave IP address. This information is irrelevant because we don't care about the IP address of the remote master. We care only that the remote master connects to the specified *slaveInterface* and sends a command to be forwarded.

The protocol type is not specified for *slaveInterface.* All messages in any Modbus IP protocol received on *slaveInterface* for *slaveStation* will be forwarded.

Serial Modbus Interface as Forwarding Interface

The *forwardIPAddress* field in the SF_TRANSLATION structure should be set to zero when the *forwardInterface* field is a Serial Modbus interface. Set forwardIPAddress to zero as follows:

```
SF_TRANSLATION sfTranslation;

sfTranslation.forwardIPAddress.s_addr = 0;
```

See Also

getSFTranslation

setSFTranslationEx

Write Store and Forward Translation method 2

Syntax

```
#include <ctools.h>
struct SFTranslationStatus setSFTranslationEx(UINT16 index,
   SF_TRANSLATION_EX * pTranslation, UCHAR* userName,
   UCHAR* password);
```

Description

The **setSFTranslationEx** function copies the structure pointed to by *pTranslation* into the store and forward translation table at the location specified by *index*. Valid values for *index* are 0 to 127. The function checks for invalid translations. The translation is always stored even if invalid.

If the *userName* parameter is non-NULL then the Store and Forward entry will be set to use authentication, with the user name set to the contents of the array pointed to by *userName* and the password set to the contents of the array pointed to by *password*. Both *userName* and *password* must point to arrays of 16 characters. User names and passwords shorter than 16 characters should be padded to 16 characters with spaces. If the userName parameter is NULL then no authentication information will be stored with the Store and Forward entry.

The SF_TRANSLATION_EX structure supports a timeout and is described in the *Structures and Types* section.

The function returns a SFTranslationStatus structure. It is described in the *Structures and Types* section. The *code* field of the structure is set to one of the following. If there is an error, the *index* field is set to the location of the translation that is not valid.

Result code	***Meaning***
SF_VALID	All translations are valid
SF_NO_TRANSLATION	The entry defines re-transmission of the same message on the same port
SF_PORT_OUT_OF_RANGE	One or both of the interfaces is not valid
SF_STATION_OUT_OF_RANGE	One or both of the stations is not valid
SF_ALREADY_DEFINED	The translation already exists in the table
SF_INDEX_OUT_OF_RANGE	The entry referenced by index does not exist in the table
SF_INVALID_FORWARDING_IP	The forwarding IP address is invalid.

Notes

The ***TeleBUS Protocols User Manual*** describes store and forward messaging mode.

Writing a translation with both stations set to station 65535 can clear a translation in the table. Station 65535 is not a valid station.

The Modbus protocol type and communication parameters may differ between serial ports. The store and forward messaging will translate the protocol messages.

Translations describe the communication path of the master command: e.g. the slave interface which receives the command and the forwarding interface to forward the command. The response to the command is automatically returned to master through the same communication path in reverse.

Do not specify additional entries in the Store and Forward Table to describe the response path.

The IO_SYSTEM resource must be requested before calling this function.

To save the Store and Forward Table with the controller settings in flash memory so that it is loaded on controller reset, call **flashSettingsSave** as shown below.

```
// save Store & Forward table with controller settings
request_resource(FLASH_MEMORY);
flashSettingsSave(CS_RUN);
release_resource(FLASH_MEMORY);
```

Translations may involve any combination of interfaces. The interfaces may be running a Serial Modbus or Modbus IP protocol.

Slave Interface	Forwarding Interface
Serial Modbus Interface: e.g. com1, com2, or com3	Serial Modbus Interface: e.g. com1, com2, or com3
Modbus IP Interface: e.g. Ethernet1	Serial Modbus Interface: e.g. com1, com2, or com3
Serial Modbus Interface: e.g. com1, com2, or com3	Modbus IP Network: e.g. Modbus/TCP, Modbus RTU over UDP, or Modbus ASCII over UDP
Modbus IP Interface: e.g. Ethernet1	Modbus IP Network: e.g. Modbus/TCP, Modbus RTU over UDP, or Modbus ASCII over UDP

Modbus IP Network as Forwarding Interface

When forwarding to a TCP or UDP network, the protocol type is selected for the *forwardInterface* in the SF_TRANSLATION_EX structure. The IP Stack automatically determines the exact interface (e.g. Ethernet1) to use when it searches the network for the *forwardIPAddress*.

Also, when forwarding on a TCP or UDP network, the forwarding destination IP address must be entered as the *forwardIPAddress*. The *forwardIPAddress* is entered as an IP address string of the format 255.255.255.255. The *forwardIPAddress* is needed to know where to connect so that the command can be forwarded to its final destination.

Modbus IP Network as Slave Interface
Note that there is no field for a slave IP address. This information is irrelevant because we don't care about the IP address of the remote master. We care only that the remote master connects to the specified *slaveInterface* and sends a command to be forwarded.

The protocol type is not specified for *slaveInterface.* All messages in any Modbus IP protocol received on *slaveInterface* for *slaveStation* will be forwarded.

Serial Modbus Interface as Forwarding Interface
The *forwardIPAddress* field in the SF_TRANSLATION_EX structure should be set to zero when the *forwardInterface* field is a Serial Modbus interface. Set forwardIPAddress to zero as follows:

```
SF_TRANSLATION_EX sfTranslation;

sfTranslation.forwardIPAddress.s_addr = 0;
```

See Also

getSFTranslationEx, checkSFTranslation, clearSFTranslation

setsockopt

Syntax

```
#include <ctools.h>
int setsockopt
(
      int socketDescriptor,
      int protocolLevel,
      int optionName,
      const char * optionValue,
      int optionLength
);
```

Function Description

setsockopt is used manipulate options associated with a socket. Options may exist at multiple protocol levels; they are always present at the uppermost “socket” level. When manipulating socket options, the level at which the option resides and the name of the option must be specified. To manipulate options at the “socket” level, *protocolLevel* is specified as SOL_SOCKET. To manipulate options at any other level, *protocolLevel* is the protocol number of the protocol that controls the option. For example, to indicate that an option is to be interpreted by the TCP protocol, *protocolLevel* is set to the TCP protocol number. The parameters *optionValuePtr* and *optionlength* are used to access option values for **setsockopt**. *optionName* and any specified options are passed un-interpreted to the appropriate protocol module for interpretation. The include file <ctools.h> contains definitions for the options described below. Most socket-level options take an int pointer for *optionValuePt*r. For **setsockopt**, the integer value pointed to by the *optionValuePtr* parameter should be non-zero to enable a boolean option, or zero if the option is to be disabled. SO_LINGER uses a struct linger parameter that specifies the desired state of the option and the linger interval (see below). struct linger is defined in <ctools.h>.

struct linger contains the following members:

l_onoff on = 1/off = 0

l_linger linger time, in seconds.

The following options are recognized at the socket level

SOL_SOCKET *protocolLevel* options

SO_DONTROUTE	Enable/disable routing bypass for outgoing messages. Default 0.
SO_KEEPALIVE	Enable/disable keep connections alive. Default 0.
SO_LINGER	Linger on close if data is present. Default is on with 60 seconds timeout.
SO_OOBINLINE	Enable/disable reception of out-of-band data in band. Default 0.
SO_REUSEADDR	Enable/disable local address reuse. Default 0 (disable).
SO_RCVLOWAT	The low water mark for receiving data.
SO_SNDLOWAT	The low water mark for sending data.
SO_R CVBUF	Set buffer size for input. Default 8192 bytes.
SO_SNDBUF	Set buffer size for output. Default 8192 bytes.

SO_REUSEADDR indicates that the rules used in validating addresses supplied in a **bind** call should allow reuse of local addresses. SO_KEEPALIVE enables the periodic transmission of messages on a connected socket. If the connected party fails to respond to these messages, the connection is considered broken. SO_DONTROUTE indicates that outgoing messages should bypass the standard routing facilities. Instead, messages are directed to the appropriate network interface according to the network portion of the destination address. SO_LINGER controls the action taken when unsent messages are queued on a socket and a close on the socket is performed. If the socket promises reliable delivery of data and SO_LINGER is set, the system will block the process on the close of the socket attempt until it is able to transmit the data or decides it is unable to deliver the information. A timeout period, termed the linger interval, is specified in the **setsockopt** call when SO_LINGER is requested. If SO_LINGER is disabled and a close on the socket is issued, the system will process the close of the socket in a manner that allows the process to continue as quickly as possible. The option SO_BROADCAST requests permission to send broadcast datagrams on the socket. With protocols that support out-of-band data, the SO_OOBINLINE option requests that out-of-band data be placed in the normal data input queue as received; it will then be accessible with recv call without the MSG_OOB flag.

SO_SNDBUF and SO_RCVBUF are options that adjust the normal buffer sizes allocated for output and input buffers, respectively. The buffer size may be increased for high-volume connections or may be decreased to limit the possible backlog of incoming data. The Internet protocols place an absolute limit of 64 Kbytes on these values for UDP and TCP sockets (in the default mode of operation).

The following options are recognized at the IP level:

IP_PROTOIP *protocolLevel* options

IP_TOS	IP type of service. Default 0.
IP_TTL	IP Time To Live in seconds. Default 64.
IP_MULTICAST_TTL	Change the default IP TTL for outgoing multicast datagrams
IP_MULTICAST_IF	Specify a configured IP address that will uniquely identify the outgoing interface for multicast datagrams sent on this socket. A zero IP address parameter indicates that we want to reset a previously set outgoing interface for multicast packets sent on that socket

The following options are recognized at the TCP level.

IP_PROTOTCP *protocolLevel* options

TCP_MAXSEG	Sets the maximum TCP segment size sent on the network. Note that the TCP_MAXSEG value is the maximum amount of data (including TCP options, but not the TCP header) that can be sent per segment to the peer., i.e the amount of user data sent per segment is the value given by the TCP_MAXSEG option minus any enabled TCP option (for example 12 bytes for a TCP time stamp option) . The TCP_MAXSEG value can be decreased or increased prior to a connection establishment, but it is not recommended to set it to a value higher than the IP MTU minus 40 bytes (for example 1460 bytes on Ethernet), since this would cause fragmentation of TCP segments. *Note: setting the TCP_MAXSEG option will inhibit the automactic computation of that value by the system based on the IP MTU (which avoids fragmentation), and will also inhibit Path Mtu Discovery.* After the connection has started, this value cannot be changed. Note also

	that the TCP_MAXSEG value cannot be set below 64 bytes. Default value is IP MTU minus 40 bytes.
TCP_NODELAY	Set this option value to a non-zero value, to disable the Nagle algorithm that buffers the sent data inside the TCP. Useful to allow client's TCP to send small packets as soon as possible (like mouse clicks). Default 0.

Parameters

socketDescriptor	The socket descriptor to set the options on.
protocolLevel	The protocol to set the option on. See below.
optionName	The name of the option to set. See below and above.
optionValuePtr	The pointer to a user variable from which the option value is set. User variable is of data type described below.
optionLength	The size of the user variable. It is the size of the option data type described below.

ProtocolLevel

SOL_SOCKET	Socket level protocol.
IP_PROTOIP	IP level protocol.
IP_PROTOTCP	TCP level protocol.

ProtocolLevel	Option Name	Option data type	Option value
SOL_SOCKET	SO_DONTROUTE	int	0 or 1
	SO_KEEPALIVE	int	0 or 1
	SO_LINGER	struct linger	
	SO_OOBINLINE	int	0 or 1
	SO_RCVBUF	unsigned long	
	SO_RCVLOWAT	unsigned long	
	SO_REUSEADDR	int	0 or 1
	SO_SNDBUF	unsigned long	
	SO_SNDLOWAT	unsigned long	
IP_PROTOIP	IP_TOS	unsigned char	
	IP_TTL	unsigned	

Protocol Level	Option Name	Option data type	Option value
		char	
	IP_MULTICAST_TTL	unsigned char	
	IP_MULTICAST_IF	struct in_addr	
IP_PROTOTCP	TCP_MAXSEG	int	
	TCP_NODELAY	int	0 or 1

Returns

0 Successful set of option

-1 An error occurred

setsockopt will fail if:

EBADF	The socket descriptor is invalid
EINVAL	One of the parameters is invalid
ENOPROTOOPT	The option is unknown at the level indicated.
EPERM	Option cannot be set after the connection has been established.
ENETDOWN	Specified interface not configured yet
EADDRINUSE	Multicast host group already added to the interface
ENOBUF	Not enough memory to add new multicast entry.
ENOENT	Attempted to delete a non-existent multicast entry on the specified interface.

setStatusBit

Set Bits in Controller Status Code

Syntax

```
#include <ctools.h>
UINT16 setStatusBit(UINT16 bitMask);
```

Description

The **setStatusBit** function sets the bits indicated by *bitMask* in the controller status code. When the status code is non-zero, the STAT LED blinks a binary sequence corresponding to the code. If *code* is zero, the STAT LED turns off.

The function returns the value of the status register.

Notes

The status output opens if *code* is non-zero. Refer to the ***System Hardware Manual*** for more information.

The binary sequence consists of short and long flashes of the error LED. A short flash of 1/10th of a second indicates a binary zero. A binary one is indicated by a longer flash of approximately 1/2 of a second. The least significant digit is output first. As few bits as possible are displayed – all leading zeros are ignored. There is a two second delay between repetitions.

The STAT LED is located on the top left hand corner of the controller board.

Bits 0, 1 and 2 of the status code are used by the controller firmware. Attempting to control these bits will result in indeterminate operation.

See Also

getStatusBit

setStatusMode

Set Source for Status LED

Syntax

```
#include <ctools.h>
void setStatusLed(UINT16 mode);
```

Description

The **setStatusMode** function controls wether `APPLICATION` or `SYSTEM` status bits are shown on the STAT LED.

The function has no return value.

setWakeSource

Sets Conditions for Waking from Sleep Mode

Syntax

```
#include <ctools.h>
void setWakeSource(UINT32 enableMask);
```

Description

The setWakeSource routine enables and disables sources that will wake up the processor. It enables all sources specified by *enableMask*. All other sources are disabled.

Valid wake up sources are listed below. Multiple sources may be ORed together.

- WAKE_SOURCE_NONE
- WAKE_SOURCE_ALL
- WAKE_SOURCE_RTC_ALARM
- WAKE_SOURCE_COUNTER_1_OVERFLOW
- WAKE_SOURCE_COUNTER_2_OVERFLOW
- WAKE_SOURCE_COUNTER_3_OVERFLOW
- WAKE_SOURCE_LED_POWER_SWITCH
- WAKE_SOURCE_DIN_1_CHANGE
- WAKE_SOURCE_COM3_VISION

Notes

Specifying WAKE_SOURCE_NONE as the wake up source will prevent the controller from waking, except by a power on reset.

See Also

getWakeSource, setPowerMode

Example

The code fragments below show how to enable and disable wake up sources.

```
/* Wake up on all sources */
setWakeSource(WAKE_SOURCE_ALL);

/* Enable wake up on real time clock only */
setWakeSource(WAKE_SOURCE_RTC_ALARM);
```

shutdown

Syntax

```
#include <ctools.h>
int shutdown
(
      int socketDescriptor,
      int howToShutdown
);
```

Function Description

Shutdown a socket in read, write, or both directions determined by the parameter *howToShutdown.*

Parameters

socketDescriptor	The socket to shutdown
howToShutdown	Direction: 0 = Read 1 = Write 2 = Both

Returns

0	Success
-1	An error occurred

shutdown will fail if:

EBADF	The socket descriptor is invalid
EINVAL	One of the parameters is invalid
ENOPROTOOPT	The option is unknown at the level indicated.

signal_event

Signal Occurrence of Event

Syntax

```
#include <ctools.h>
void signal_event(UINT32 event_number);
```

Description

The **signal_event** function signals that the *event_number* event has occurred.

If there are tasks waiting for the event, the highest priority task is made ready to execute. Otherwise the event flag is incremented. Up to 32767 occurrences of an event will be recorded. The current task is blocked if there is a higher priority task waiting for the event.

Notes

Refer to the **Real Time Operating System** section for more information on events.

Valid events are numbered 0 to RTOS_EVENTS - 1. Any events defined in ctools.h are not valid events for use in an application program.

This function can be called from application and interrupt code.

See Also

poll_event

Example

This program creates a task to wait for an event, then signals the event.

```
#include <ctools.h>

void task1(void)
{
       while(TRUE)
       {
          wait_event(20);
          fprintf(com1,"Event 20 occurred\r\n");
       }
}

int main(void)
{
       UINT32 startTime;
       create_task(task1, 75, APPLICATION, 4);

       while(TRUE)
       {
          /* body of main task loop */
          /* The body of this main task is intended         solely for
       signaling the event waited for by       task1. Normally main would
       be busy with more
          important things to do otherwise the code in
          task1 could be executed within main's wait
          loop */
```

```
            startTime = readStopwatch();
            while ((readStopwatch() – startTime) < 1000)             /*
wait for 1 s */
            {
               /* Allow other tasks to execute */
               release_processor();
            }
            signal_event(20);
         }
}
```

sleep_processor

Release Processor to other Tasks for a certain time

Syntax

```
#include <ctools.h>
void sleep_processor(UINT32 msTime);
```

Description

The **sleep_processor** function releases control of the CPU to other tasks for a certain time. Other tasks of the same priority get a chance to run, or when no such task is in a ready state lower priority tasks will run. This function is similar to **release_processor** with the difference that the CPU is released for at least msTime, which represents milliseconds. Tasks of the same priority run in a round-robin fashion, as each releases the processor to the next.

Notes

The call **sleep_processor(0)** has the same effect as the call **release_processor**.

Internally the sleep time msTime will be converted into ticks. With a 60 Hz system clock, the minimum wait time is 16.6 ms. Wait times will be rounded up to the next tick value.

Refer to the **Real Time Operating System** section for more information on tasks and task scheduling.

See Also

release_processor

sleepMode

Suspend Controller Operation

Syntax

```
#include <ctools.h>
UINT16 sleepMode(void);
```

Description

The **sleepMode** function puts the controller into a sleep mode. Sleep mode reduces the power consumption to a minimum by halting the microprocessor clock. All programs halt until the controller resumes execution. All output points turn off while the controller is in sleep mode.

The controller resumes execution under the conditions shown in the table below. The application program may disable some wake up conditions. If a wake up condition is disabled the controller will not resume execution when the condition occurs. All wake up conditions will be enabled by default. Refer to the description of the **setWakeSource** function for details.

sleepMode returns the source that woke the controller from sleep.

See Also

getWakeSource, setWakeSource

socket

Syntax

```
#include <ctools.h>
int socket
(
      int family,
      int type,
      int protocol
);
```

Function Description

socket creates an endpoint for communication and returns a descriptor. The *family* parameter specifies a communications domain in which communication will take place; this selects the protocol family that should be used. The protocol family is generally the same as the address family for the addresses supplied in later operations on the socket. These families are defined in the include file <ctools.h>. If *protocol* has been specified, but no exact match for the tuplet family, type, and protocol is found, then the first entry containing the specified family and type with zero for protocol will be used. The currently understood format is PF_INET for ARPA Internet protocols. The socket has the indicated *type*, which specifies the communication semantics.

Currently defined types are:

SOCK_STREAM

SOCK_DGRAM

SOCK_RAW

A SOCK_STREAM type provides sequenced, reliable, two-way connection-based byte streams. An out-of-band data transmission mechanism is supported. A SOCK_DGRAM socket supports datagrams (connectionless, unreliable messages of a fixed (typically small) maximum length); a SOCK_DGRAM user is required to read an entire packet with each **recv** call or variation of **recv** call, otherwise an error code of EMSGSIZE is returned. protocol specifies a particular protocol to be used with the socket. Normally only a single protocol exists to support a particular socket type within a given protocol family. However, multiple protocols may exist, in which case, a particular protocol must be specified in this manner.

The protocol number to use is particular to the “communication domain” in which communication is to take place. If the caller specifies a protocol, then it will be packaged into a socket level option request and sent to the underlying protocol layers. Sockets of type SOCK_STREAM are full-duplex byte streams. A stream socket must be in a connected state before any data may be sent or received on it. A connection to another socket is created with **connect** on the client side. On the server side, the server needs to call **listen** and then **accept**. Once connected, data may be transferred using **recv** and **send** calls or some variant of the **send** and **recv** calls. When a session has been completed, a **close** of the socket should be performed. The communications protocols used to implement a SOCK_STREAM ensure that data is not lost or duplicated. If a piece of data (for which the peer protocol has buffer space) cannot be successfully transmitted within a reasonable length of time, then the connection is considered broken and calls will indicate an error with (-1) return value and with ETIMEDOUT as the specific socket error. The TCP protocols optionally keep sockets “warm” by forcing transmissions roughly every two hours in the absence of other activity. An error is then indicated if no response can be elicited on an otherwise idle connection for an extended period (for instance 5 minutes). SOCK_DGRAM or SOCK_RAW sockets allow datagrams to be sent to

correspondents named in **sendto** calls. Datagrams are generally received with **recvfrom** which returns the next datagram with its return address. The operation of sockets is controlled by socket level options. These options are defined in the file <ctools.h>. **setsockopt** and **getsockopt** are used to set and get options, respectively.

Parameters

family The protocol family to use for this socket (currently only PF_INET is used).

type The type of socket.

protocol The layer 4 protocol to use for this socket.

Family	***Type***	***Protocol***	***Actual protocol***
PF_INET	SOCK_DGRAM	IPPROTO_UDP	UDP
PF_INET	SOCK_STREAM	IPPROTO_TCP	TCP
PF_INET	SOCK_RAW	IPPROTO_ICMP	ICMP
PF_INET	SOCK_RAW	IPRPTOTO_IGMP	IGMP.

Returns

New Socket Descriptor or –1 on error.

If an error occurred, the socket error can be retrieved by calling **getErrorCode**(socketDescriptor).

socket will fail if:

EMFILE	No more sockets are available
ENOBUFS	There was insufficient user memory available to complete the operation
EPROTONOSUPPORT	The protocol type or the specified protocol is not supported within this family.

start_protocol

Start Serial Protocol

Syntax

```
#include <ctools.h>
INT16 start_protocol(UCHAR port);
```

Description

The **start_protocol** function enables a protocol on the specified serial port. It returns TRUE if the protocol was enabled and FALSE if it was not. The protocol settings of the specified serial port determine the protocol type enabled by this function.

This function should only be needed in the context of the startup function **appstart**.

See Also

set_port, get_port

startup_task

Identify Start Up Task

Syntax

```
#include <ctools.h>
void *startup_task(void);
```

Description

The **startup_task** function returns the address of the system or application start up task.

Notes

This function is used by the reset routine. It is normally not used in an application program.

startTimedEvent

Enable Signaling of a Regular Event

Syntax

```
#include <ctools.h>
UINT16 startTimedEvent(UINT16 event, UINT16 interval);
```

Description

The **startTimedEvent** function causes the specified *event* to be signaled at the specified *interval*. *interval* is measured in multiples of 0.1 seconds. The task that is to receive the events should use the **wait_event** or **poll_event** functions to detect the event.

The function returns TRUE if the event can be signaled. If interval is 0 or if the event number is not valid, the function returns FALSE and no change is made to the event signaling (a previously enabled event will not be changed).

Notes

Valid events are numbered 0 to RTOS_EVENTS - 1. Any events defined in primitiv.h are not valid events for use in an application program.

The application program should stop the signaling of timed events when the task which waits for the events is ended. If the event signaling is not stopped, events will continue to build up in the queue until a function waits for them. The example below shows a simple method using the **installExitHandler** function.

See Also

endTimedEvent

wait_event

Wait for an Event

Syntax

```
#include <ctools.h>
void wait_event(UINT32 event);
```

Description

The **wait_event** function tests if an event has occurred. If the event has occurred, the event counter is decrements and the function returns. If the event has not occurred, the task is blocked until it does occur.

Notes

Refer to the **Real Time Operating System** section for more information on events.

Valid events are numbered 0 to RTOS_EVENTS - 1. Any events defined in primitiv.h are not valid events for use in an application program.

Example

See the example for the **signal_event** function.

wd_auto

Automatic Watchdog Timer Mode

Syntax

```
#include <ctools.h>
void wd_auto(void);
```

Description

The **wd_auto** function gives control of the watchdog timer to the operating system. The timer is automatically updated by the system.

Notes

Refer to the **Functions Overview** section for more information.

Example

See the example for the **wd_manual** function

wd_enabled

Enable Watchdog

Syntax

```
#include <ctools.h>
void wd_enabled( BOOLEAN state);
```

Description

The function **wd_enabled** enables or disables the controller watchdog. This function should only be needed in the context of the startup function **appstart**, where it is called only when a debug build is made of the application.

By default a Release build of the application enables the watchdog and a Debug build of the application disables the watchdog.

The watchdog must be disabled in order to debug an application using the source-level debugging (e.g. stepping, setting breakpoints) tools provided by the Hitachi HDI and Emulator.

Calling the function with *state* set to TRUE enables the watchdog. Calling the function with *state* set to FALSE disables the watchdog.

wd_manual

Manual Watchdog Timer Mode

Syntax

```
#include <ctools.h>
void wd_manual(void);
```

Description

The **wd_manual** function takes control of the watchdog timer.

Notes

The application program must retrigger the watchdog timer at least every 0.5 seconds using the **wd_pulse** function, to prevent an controller reset.

Refer to the **Functions Overview** section for more information.

See Also

wd_enabled

Example

This program takes control of the watchdog timer for a critical section of code, then returns it to the control of the operating system.

```
#include <ctools.h>

int main(void)
{
          wd_manual();
          wd_pulse();
          /* ... code executing in less than 0.5 s */
          wd_pulse();
          /* ... code executing in less than 0.5 s */
          wd_auto()
          /* ... as much code as you wish */
}
```

wd_pulse

Retrigger Watchdog Timer

Syntax

```
#include <ctools.h>
void wd_pulse(void);
```

Description

The **wd_pulse** function retriggers the watchdog timer.

Notes

The **wd_pulse** function must execute at least every 0.5 seconds, to prevent a controller reset, if the **wd_manual** function has been executed.

Refer to the **Functions Overview** section for more information.

Example

See the example for the **wd_manual** function

writeBoolVariable

Write to ISaGRAF Boolean Variable (ISaGRAF firmware only)

Syntax

```
#include <ctools.h>
BOOLEAN writeBoolVariable(UCHAR * varName, UCHAR value)
```

Description

This function writes to the specified boolean variable.

The variable is specified by its name expressed as a character string. The name is case insensitive (The ISaGRAF Dictionary also treats variable names as case insensitive). If the variable is found, TRUE is returned and the specified *value* is written to the variable. If the variable is not found or if the ISaGRAF Symbols Status is invalid, nothing is done and FALSE is returned. The ISaGRAF Symbols Status is invalid if the Application TIC code download and Application Symbols download do not share the same symbols CRC checksum.

TRUE is written when *value* is any non-zero value. FALSE is written when *value* is 0.

Notes

This function requires the ISaGRAF Application Symbols to be downloaded to the controller in addition to the Application TIC code. This function provides a convenient method to access ISaGRAF variables by name; however, because the variable name must be looked up in the ISaGRAF variable list each call, the performance of the function may be slow for large numbers of variables. For better performance, use the variable's network address and the **setdbase** function.

The IO_SYSTEM system resource must be requested before calling this function.

Example

This program writes a TRUE state to the boolean variable named "Switch1".

```
#include <ctools.h>

int main(void)
{
   BOOLEAN       status;

   request_resource(IO_SYSTEM);
   status = writeBoolVariable("Switch1", TRUE);
   release_resource(IO_SYSTEM);
}
```

writeIntVariable

Write to ISaGRAF Integer Variable (ISaGRAF firmware only)

Syntax

```
#include <ctools.h>
BOOLEAN writeIntVariable(UCHAR * varName, INT32 long value)
```

Description

This function writes to the specified integer variable.

The variable is specified by its name expressed as a character string. The name is case insensitive (The ISaGRAF Dictionary also treats variable names as case insensitive). If the variable is found, TRUE is returned and the specified signed long *value* is written to the variable. If the variable is not found or if the ISaGRAF Symbols Status is invalid, nothing is done and FALSE is returned. The ISaGRAF Symbols Status is invalid if the Application TIC code download and Application Symbols download do not share the same symbols CRC checksum.

Notes

This function requires the ISaGRAF Application Symbols to be downloaded to the controller in addition to the Application TIC code. This function provides a convenient method to access ISaGRAF variables by name; however, because the variable name must be looked up in the ISaGRAF variable list each call, the performance of the function may be slow for large numbers of variables. For better performance, use the variable's network address and the **setdbase** function.

The IO_SYSTEM system resource must be requested before calling this function.

Example

This program writes the value 120,000 to the integer variable named "Pressure1".

```
#include <ctools.h>

int main(void)
{
   BOOLEAN       status;

   request_resource(IO_SYSTEM);
   status = writeIntVariable("Pressure1", 120000);
   release_resource(IO_SYSTEM);
}
```

writeRealVariable

Write to ISaGRAF Real Variable (ISaGRAF firmware only)

Syntax

```
#include <ctools.h>
BOOLEAN writeRealVariable(UCHAR * varName, float value)
```

Description

This function writes to the specified real (i.e. floating point) variable.

The variable is specified by its name expressed as a character string. The name is case insensitive (The ISaGRAF Dictionary also treats variable names as case insensitive). If the variable is found, TRUE is returned and the specified floating-point *value* is written to the variable. If the variable is not found or if the ISaGRAF Symbols Status is invalid, nothing is done and FALSE is returned. The ISaGRAF Symbols Status is invalid if the Application TIC code download and Application Symbols download do not share the same symbols CRC checksum.

Notes

This function requires the ISaGRAF Application Symbols to be downloaded to the controller in addition to the Application TIC code. This function provides a convenient method to access ISaGRAF variables by name; however, because the variable name must be looked up in the ISaGRAF variable list each call, the performance of the function may be slow for large numbers of variables. For better performance, use the variable's network address and the **setdbase** function.

The IO_SYSTEM system resource must be requested before calling this function.

Example

This program writes the value 25.607 to the real variable named "Flowrate".

```
#include <ctools.h>

int main(void)
{
   BOOLEAN       status;

   request_resource(IO_SYSTEM);
   status = writeRealVariable("Flowrate", 25.607);
   release_resource(IO_SYSTEM);
}
```

writeMsgVariable

Write to ISaGRAF Message Variable (ISaGRAF firmware only)

Syntax

```
#include <ctools.h>
BOOLEAN writeMsgVariable(UCHAR * varName, UCHAR * msg)
```

Description

This function writes to the specified message variable.

The variable is specified by its name expressed as a character string. The name is case insensitive (The ISaGRAF Dictionary also treats variable names as case insensitive). If the variable is found, TRUE is returned and the specified string is written to the message variable. If the variable is not found or if the ISaGRAF Symbols Status is invalid, nothing is done and FALSE is returned. The ISaGRAF Symbols Status is invalid if the Application TIC code download and Application Symbols download do not share the same symbols CRC checksum.

The pointer *msg* must point to a character string large enough to hold the maximum length declared for the specified message variable plus two length bytes and a null termination byte (i.e. max declared length + 3).

When writing to the message variable, all bytes are copied except the first byte (max length byte) and the last byte (null termination byte). ISaGRAF message variables have the following format:

Byte Location	***Description***
0	Maximum length as declared in ISaGRAF Dictionary (1 to 255)
1	Current Length = location of first null byte (0 to maximum length)
2	First message data byte
…	
max + 1	Last byte in message buffer
max + 2	Null termination byte (Terminates a message having the maximum length.)

Notes

This function requires the ISaGRAF Application Symbols to be downloaded to the controller in addition to the Application TIC code. This function provides a convenient method to access ISaGRAF variables by name; however, because the variable name must be looked up in the ISaGRAF variable list each call, the performance of the function may be slow for large numbers of variables. For better performance, use the variable's network address and the **setdbase** function.

The IO_SYSTEM system resource must be requested before calling this function.

Example

This program writes the message “Warning” to the message variable named “TextData”. TextData has a maximum length of 10 bytes and a current length of 7 bytes.

```
#include <ctools.h>

int main(void)
{
   BOOLEAN       status;
   unsigned char msg[13];

   msg[0] = 10;
   msg[1] = 7;
   msg[2] = 'W';
   msg[3] = 'a';
   msg[4] = 'r';
   msg[5] = 'n';
   msg[6] = 'i';
   msg[7] = 'n';
   msg[8] = 'g';
   msg[9] =  0;
   msg[10] = 0;
   msg[11] = 0;
   msg[12] = 0;

   request_resource(IO_SYSTEM);
   status = writeMsgVariable("TextData", msg);
   release_resource(IO_SYSTEM);
}
```

writeTimerVariable

Write to ISaGRAF Timer Variable (ISaGRAF firmware only)

Syntax

```
#include <ctools.h>
BOOLEAN writeTimerVariable(UCHAR * varName, UINT32 value)
```

Description

This function writes a value in milliseconds to the specified timer variable. The maximum value that may be written is 86399999 ms (or 24 hours). If the *value* is greater than 86399999 ms, the *value* modulus 86399999 is written to the timer variable. The specified timer may be active or stopped.

The variable is specified by its name expressed as a character string. The name is case insensitive (The ISaGRAF Dictionary also treats variable names as case insensitive). If the variable is found, TRUE is returned and the specified unsigned long *value* is written to the variable. If the variable is not found or if the ISaGRAF Symbols Status is invalid, nothing is done and FALSE is returned. The ISaGRAF Symbols Status is invalid if the Application TIC code download and Application Symbols download do not share the same symbols CRC checksum.

Notes

This function requires the ISaGRAF Application Symbols to be downloaded to the controller in addition to the Application TIC code. This function provides a convenient method to access ISaGRAF variables by name; however, because the variable name must be looked up in the ISaGRAF variable list each call, the performance of the function may be slow for large numbers of variables. For better performance, use the variable's network address and the **setdbase** function.

The IO_SYSTEM system resource must be requested before calling this function.

Example

This program writes the value 10000 ms to the timer variable named "Delay".

```
#include <ctools.h>

int main(void)
{
   BOOLEAN       status;

   request_resource(IO_SYSTEM);
   status = writeTimerVariable("Delay", 10000);
   release_resource(IO_SYSTEM);
}
```

Macro Definitions

A

Macro	*Definition*
AD_BATTERY	Internal AD channel connected to lithium battery
AD_THERMISTOR	Internal AD channel connected to thermistor
ADDITIVE	Additive checksum
AIN_END	Number of last analog input channel.
AIN_START	Number of first analog input channel.
AIO_BADCHAN	Error code: bad analog input channel specified.
AIO_SUPPORTED	If defined indicates analog I/O supported.
AIO_TIMEOUT	Error code: input device did not respond.
AO	Variable name: alarm output address
AOUT_END	Number of last analog output channel.
AOUT_START	Number of first analog output channel.
APPLICATION	Specifies an application type task. All application tasks are terminated by the end_application function.
AT_ABSOLUTE	Specifies a fixed time of day alarm.
AT_NONE	Disables alarms

B

Macro	*Definition*
BACKGROUND	System event: background I/O requested. The background I/O task uses this event. It should not be used in an application program.
BASE_TYPE_MASK	Controller type bit mask
BAUD110	Specifies 110-baud port speed.
BAUD115200	Specifies 115200-baud port speed.
BAUD1200	Specifies 1200-baud port speed.
BAUD150	Specifies 150-baud port speed.
BAUD19200	Specifies 19200-baud port speed.
BAUD2400	Specifies 2400-baud port speed.
BAUD300	Specifies 300-baud port speed.
BAUD38400	Specifies 38400-baud port speed.
BAUD4800	Specifies 4800-baud port speed.
BAUD57600	Specifies 57600-baud port speed.
BAUD600	Specifies 600-baud port speed.
BAUD75	Specifies 75-baud port speed.
BAUD9600	Specifies 9600-baud port speed.
BYTE_EOR	Byte-wise exclusive OR checksum

C

Macro	*Definition*
CA	Variable name: cascade setpoint source
CLASS0_FLAG	Specifies a flag for enabling DNP Class 0 data
CLASS1_FLAG	Specifies a flag for enabling DNP Class 1 data
CLASS2_FLAG	Sspecifies a flag for enabling DNP Class 2 data
CLASS3_FLAG	Specifies a flag for enabling DNP Class 3 data
CLOSED	Specifies switch is in closed position
COLD_BOOT	Cold-boot switch depressed when CPU was reset.
com1	Points to a file object for the com1 serial port.
COM1_RCVR	System event: indicates activity on com1 receiver. The meaning depends on the character handler installed.
com2	Points to a file object for the com2 serial port.
COM2_RCVR	System event: indicates activity on com2 receiver. The meaning depends on the character handler installed.
com3	Points to a file object for the com3 serial port.
COM3_RCVR	System event: indicates activity on com3 receiver. The meaning depends on the character handler installed.
COUNTER_CHANNELS	Specifies number of 5000 Series counter input channels
COUNTER_END	Number of last counter input channel
COUNTER_START	Number of first counter input channel
COUNTER_SUPPORTED	If defined indicates counter I/O hardware supported.
CPU_CLOCK_RATE	Frequency of the system clock in cycles per second
CR	Variable name: control register
CRC_16	CRC-16 type CRC checksum (reverse algorithm)
CRC_CCITT	CCITT type CRC checksum (reverse algorithm)

D

Macro	*Definition*
DATA_SIZE	Maximum length of the HART command or response field.
DATA7	Specifies 7 bit world length.
DATA8	Specifies 8 bit word length.
DB	Variable name: deadband
DB_BADSIZE	Error code: out of range address specified
DB_BADTYPE	Error code: bad database addressing type specified
DB_OK	Error code: no error occurred
DE_BadConfig	The modem configuration structure contains an error
DE_BusyLine	The phone number called was busy
DE_CallAborted	A call in progress was aborted by the user
DE_CarrierLost	The connection to the remote site was lost (modem reported NO CARRIER). Carrier is lost for a time exceeding the S10 setting in the modem. Phone lines with call waiting are very susceptible to this condition.
DE_FailedToConnect	The modem could not connect to the remote site
DE_InitError	Modem initialization failed (the modem may be turned off)

Macro	*Definition*
DE_NoDialTone	Modem did not detect a dial tone or the S6 setting in the modem is too short.
DE_NoError	No error has occurred
DE_NoModem	The serial port is not configured as a modem (port type must be RS232_MODEM). Or no modem is connected to the controller serial port.
DE_NotInControl	The serial port is in use by another modem function or has answered an incoming call.
DIN_END	Number of last regular digital input channel.
DIN_START	Number of first regular digital input channel
DIO_SUPPORTED	If defined indicates digital I/O hardware supported.
DISABLE	Specifies flow control is disabled.
DNP	Specifies the DNP protocol for the serial port
DO	Variable name: decrease output
DOUT_END	Number of last regular digital output channel.
DOUT_START	Number of first regular digital output channel
DS_Calling	The controller is making a connection to a remote controller
DS_Connected	The controller is connected to a remote controller
DS_Inactive	The serial port is not in use by a modem
DS_Terminating	The controller is ending a connection to a remote controller.
DYNAMIC_MEMORY	System resource: all memory allocation functions such as `malloc` and `alloc`.

E

Macro	*Definition*
ENABLE	Specifies flow control is enabled.
ER	Variable name: error
EVEN	Specifies even parity.
EX	Variable name: automatic execution period
EXTENDED_DIN_END	Number of last extended digital input channel.
EXTENDED_DIN_START	Number of first extended digital input channel
EXTENDED_DOUT_END	Number of last extended digital output channel.
EXTENDED_DOUT_START	Number of first extended digital output channel

F

Macro	Definition
FOPEN_MAX	Redefinition of macro from stdio.h
FORCE_MULTIPLE_COILS	Modbus function code
FORCE_SINGLE_COIL	Modbus function code
FULL	Specifies full duplex.

G

Macro	*Definition*
GASFLOW	Gas Flow calculation firmware option

H

Macro	*Definition*
HALF	Specifies half duplex.
HT_5209	Specifies that 5209 hardware is persent

I

Macro	*Definition*
IO_SYSTEM	System resource for all I/O hardware functions.

L

Macro	*Definition*
LAN_ENABLED	Enables LAN communication
LAN_DISABLED	Disables LAN communication, reducing power consumption.
LED_OFF	Specifies LED is to be turned off.
LED_ON	Specifies LED is to be turned on.
LINEAR	Specifies linear database addressing.
LOAD_MULTIPLE_REGISTERS	Modbus function code
LOAD_SINGLE_REGISTER	Modbus function code
LOW_POWER_MODE	Reduces the operating speed of the controller, reducing power consumption.

M

Macro	*Definition*
MAX_NUMBER_OF_LOGS	The maximum number of data logs.
MAX_NUMBER_OF_FIELDS	The maximum number of fields in a data log record.
MAX_PRIORITY	The maximum task priority.
MM_BAD_ADDRESS	Master message status: invalid database address
MM_BAD_FUNCTION	Master message status: invalid function code
MM_BAD_LENGTH	Master message status: invalid message length
MM_BAD_SLAVE	Master message status: invalid slave station address
MM_NO_MESSAGE	Master message status: no message was sent.
MM_PROTOCOL_NOT_SUPPORTED	Master message status: selected protocol is not supported.
MM_RECEIVED	Master message status: response was received.
MM_SENT	Master message status: message was sent.
MODBUS	Specifies Modbus database addressing.
MM_EXCEPTION_FUNCTION	Master message status: Modbus slave returned a function exception.
MM_EXCEPTION_ADDRESS	Master message status: Modbus slave returned an

Macro	*Definition*
	address exception.
MM_EXCEPTION_VALUE	Master message status: Modbus slave returned a value exception.
MODBUS_ASCII	Specifies the Modbus ASCII protocol emulation for the serial port.
MODBUS_PARSER	System resource: Modbus protocol message parser.
MODBUS_RTU	Specifies the Modbus RTU protocol emulation for the serial port.
MODEM_CMD_MAX_LEN	Maximum length of the modem initialization command string
MODEM_MSG	System event: new modem message generated.
MSG_DATA	Specifies the data field in an envelope contains a data value.
MSG_POINTER	Specifies the data field in an envelope contains a pointer.
MT_Ain4	Four channel analog input module
MT_Ain8	Eight channel analog input module
MT_Aout2	Two channel analog output module
MT_Aout4	Four channel analog output module
MT_Din8	Eight channel digital input module
MT_Din16	Sixteen channel digital input module
MT_Dout8	Eight channel digital output module
MT_Dout16	Sixteen channel digital output module
MT_Counter4	Four channel counter input module
MT_5601Inputs	5601 module analog and digital inputs
MT_5601Outputs	5601 module digital outputs
MT_5904Inputs	HART interface inputs
MT_5904Outputs	HART interface outputs
MT_CounterSP2	SCADAPack2 controller board counter inputs
MT_SP2Inputs	SCADAPack2 controller board inputs
MT_SP2Outputs	SCADAPack2 controller board outputs
MT_Dout32	Thirty two channel digital output module
MT_Din32	Thirty two channel digital input module
MT_5604Inputs	5604 module analog and digital inputs
MT_5604Outputs	5604 module digital outputs
MT_Aout4_Checksum	Four channel analog output module with checksum. This module type can only be used with analog output modules with checksum support.

N

Macro	*Definition*
NEVER	System event: this event will never occur.
NEW_PROGRAM	Application program is newly loaded.
NO_ERROR	Error code: indicates no error has occurred.
NO_PROTOCOL	Specifies no communication protocol for the serial port.
NONE	Specifies no parity.
NORMAL	Specifies the normal Modbus response type code for a Modbus Handler
NORMAL_POWER_MODE	Sets the controller to run a full operating speed.

Macro	*Definition*
NOTYPE	Specifies serial port type is not known.
NUMAB	Number of registers in the Allan-Bradley database.
NUMCOIL	Number of registers in the Modbus coil section.
NUMHOLDING	Number of registers in the Modbus holding register section.
NUMINPUT	Number of registers in the Modbus input register section.
NUMLINEAR	Number of registers in the linear database.
NUMSTATUS	Number of registers in the Modbus status section.

O

Macro	*Definition*
ODD	Specifies odd parity.
OPEN	Specifies switch is in open position

P

Macro	*Definition*
PC_FLOW_RX_RECEIVE_STOP	Receiver disabled after receipt of a message.
PC_FLOW_RX_XON_XOFF	Receiver Xon/Xoff flow control.
PC_FLOW_TX_IGNORE_CTS	Transmitter flow control ignores CTS.
PC_FLOW_TX_XON_XOFF	Transmitter Xon/Xoff flow control.
PC_PROTOCOL_RTU_FRAMING	Modbus RTU framing.
PHONE_NUM_MAX_LEN	Maximum length of the phone number string
PM_CPU_FULL_CLOCK	The CPU is set to run at full speed
PM_CPU_REDUCED_CLOCK	The CPU is set to run at a reduced speed
PM_CPU_SLEEP	The CPU is set to sleep mode
PM_LAN_ENABLED	The LAN is enabled
PM_LAN_DISABLED	The LAN is disabled
PM_USB_PERIPHERAL_ENABLED	The USB peripheral port is enabled
PM_USB_PERIPHERAL_DISABLED	The USB peripheral port is disabled
PM_USB_HOST_ENABLED	The USB host port is enabled
PM_USB_HOST_DISABLED	The USB host port is disabled
PM_UNAVAILABLE	The status of the device could not be read.
PM_NO_CHANGE	The current value will be used
PROGRAM_EXECUTED	Application program has been executed.
PROGRAM_NOT_LOADED	The requested application program is not loaded.

R

Macro	*Definition*
READ_COIL_STATUS	Modbus function code
READ_EXCEPTION_STATUS	Modbus function code
READ_HOLDING_REGISTER	Modbus function code
READ_INPUT_REGISTER	Modbus function code
READ_INPUT_STATUS	Modbus function code

Macro	*Definition*
READSTATUS	enum ReadStatus
REPORT_SLAVE_ID	Modbus function code
RFC_MODBUS_RTU	Flow control type, may be used in place of ENABLE
RFC_NONE	Flow control type, may be used in place of DISABLE
RS232	Specifies serial port is an RS-232 port.
RS232_MODEM	Specifies serial port is an RS-232 dial-up modem.
RS485_2WIRE	Specifies serial port is a 2 wire RS-485 port.
RTOS_ENVELOPES	Number of RTOS envelopes.
RTOS_EVENTS	Number of RTOS events.
RTOS_PRIORITIES	Number of RTOS task priorities.
RTOS_RESOURCES	Number of RTOS resource flags.
RTOS_TASKS	Number of RTOS tasks.
RUN	Run/Service switch is in RUN position.

S

Macro	*Definition*
S_MODULE_FAILURE	Status LED code for I/O module communication failure
S_NORMAL	Status LED code for normal status
SERIAL_PORTS	Number of serial ports.
SERVICE	Run/Service switch is in SERVICE position.
SF_ALREADY_DEFINED	Result code: translation is already defined in the table
SF_INDEX_OUT_OF_RANGE	Result code: invalid translation table index
SF_NO_TRANSLATION	Result code: entry does not define a translation
SF_PORT_OUT_OF_RANGE	Result code: serial port is not valid
SF_STATION_OUT_OF_RANGE	Result code: station number is not valid
SF_TABLE_SIZE	Number of entries in the store and forward table
SF_VALID	Result code: translation is valid
SIGNAL_CTS	I/O line bit mask: clear to send signal
SIGNAL_CTS	Matches status of CTS input.
SIGNAL_DCD	I/O line bit mask: carrier detect signal
SIGNAL_DCD	Matches status of DCD input.
SIGNAL_OFF	Specifies a signal is de-asserted
SIGNAL_OH	I/O line bit mask: off hook signal
SIGNAL_OH	Not supported – forced low (1).
SIGNAL_ON	Specifies a signal is asserted
SIGNAL_RING	I/O line bit mask: ring signal
SIGNAL_RING	Not supported – forced low (0).
SIGNAL_VOICE	I/O line bit mask: voice/data switch signal
SIGNAL_VOICE	Not supported – forced low (0).
SLEEP_MODE_SUPPORTED	Defined if sleep function is supported
SMARTWIRE_5201_5202	SmartWIRE 5201 and 5202 controllers
STACK_SIZE	Size of the machine stack.
START_COIL	Start of the coils section in the linear database.
START_HOLDING	Start of the holding register section in the linear database.
START_INPUT	Start of the input register section in the linear database.
START_STATUS	Start of the status section in the linear database.

Macro	*Definition*
STARTUP_ APPLICATION	Specifies the application start up task.
STARTUP_SYSTEM	Specifies the system start up task.
STOP1	Specifies 1 stop bit.
SYSTEM	Specifies a system type task. System tasks are not terminated by the end_application function.

T

Macro	*Definition*
T_CELSIUS	Specifies temperatures in degrees Celsius
T_FAHRENHEIT	Specifies temperatures in degrees Fahrenheit
T_KELVIN	Specifies temperatures in degrees Kelvin
T_RANKINE	Specifies temperatures in degrees Rankine
TELESAFE_6000_16EX	TeleSAFE 6000-16EX controller
TELESAFE_MICRO_16	TeleSAFE Micro16 controller
TFC_IGNORE_CTS	Flow control type, may be used in place of ENABLE
TFC_NONE	Flow control type, may be used in place of DISABLE
TIMER_BADINTERVAL	Error code: invalid timer interval
TIMER_BADTIMER	Error code: invalid timer
TIMER_MAX	Number of last valid software timer.
TS_EXECUTING	Task status indicating task is executing.
TS_READY	Task status indicating task is ready to execute
TS_WAIT_ RESOURCE	Task status indicating task is blocked waiting for a resource
TS_WAIT_ENVELOPE	Task status indicating task is blocked waiting for an envelope
TS_WAIT_EVENT	Task status indicating task is blocked waiting for an event
TS_WAIT_MESSAGE	Task status indicating task is blocked waiting for a message

V

Macro	*Definition*
VI_DATE_SIZE	Number of characters in version information date field

W

Macro	*Definition*
WRITESTATUS	enum WriteStatus
WAKE_SOURCE_NONE	Bit mask to disable all wake sources
WAKE_SOURCE_RTC_ALARM	Bit mask to enable real time clock as a wake up source
WAKE_SOURCE_COUNTER_1_OVERFLOW	Bit mask to enable counter 1 overflow as a wake up source
WAKE_SOURCE_COUNTER_2_OVERFLOW	Bit mask to enable counter 2 overflow as a wake up source

Macro	*Definition*
WAKE_SOURCE_COUNTER_3_OVERFLOW	Bit mask to enable counter 3 overflow as a wake up source
WAKE_SOURCE_LED_POWER_SWITCH	Bit mask to enable LED power switch as a wake up source
WAKE_SOURCE_DIN_1_CHANGE	Bit mask to enable DIN 1 change of state as a wake up source
WAKE_SOURCE_COM3_VISION	Bit mask to enable the SCADAPack Vision on COM 3 as a wake up source
WAKE_SOURCE_ALL	Bit mask to enable all wake up sources

Structures and Types

ADDRESS_MODE

The ADDRESS_MODE enumerated type describes addressing modes for communication protocols.

```
typedef enum addressMode_t
      {
      AM_standard = 0,
      AM_extended
      }
      ADDRESS_MODE;
```

- `AM_standard` returns standard Modbus addressing. Standard addressing allows 255 stations and is compatible with standard Modbus devices
- `AM_extended` returns extended addressing. Extended addressing allows 65534 stations.

ALARM_SETTING

The ALARM_SETTING structure defines a real time clock alarm setting.

```
typedef struct alarmSetting_tag {
      UINT16 type;
      UINT16 hour;
      UINT16 minute;
      UINT16 second;
      } ALARM_SETTING;
```

- `type` specifies the type of alarm. It may be the AT_NONE or AT_ABSOLUTE macro.
- `hour` specifies the hour at which the alarm will occur.
- `minute` specifies the minute at which the alarm will occur.
- `second` specifies the second at which the alarm will occur.

COM_INTERFACE

The COM_INTERFACE enumerated type defines a communication interface type and may have one of the following values.

```
      typedef enum interface_t
      {
         CIF_Com1         = 1,
         CIF_Com2         = 2,
         CIF_Com3         = 3,
         CIF_Ethernet1    = 100
      }
      COM_INTERFACE;
```

COMM_ENDPOINT

The COMM_ENDPOINT structure defines a communication endpoint. If ethernet based protocols are not used then the ipAddress, and portNumber fields should be set to 0.

```
struct
{
      COM_INTERFACE     interface;
      UINT32            stationAddress;
      UINT32            ipAddress;
      UINT16            portNumber;
      UCHAR             protocol;
}
COMM_ENDPOINT;
```

CONNECTION_TYPE

The CONNECTION_TYPE enumerated type defines connection types supported by the connection pool.

```
typedef enum ipConnection_t
{
   CT_Unused = 0,
   CT_Slave,           // slave task connection
   CT_MasterISaGRAF,   // master task connection created for an
                          ISaGRAF masterip FB
   CT_MasterCApp,      // master task connection created for a
                          C++ application
   CT_MasterSF         // master task connection created for store
                          and forward
}
CONNECTION_TYPE;
```

Only the connection type CT_MasterCApp may be used in C++ applications.

DATALOG_CONFIGURATION

The data log configuration structure holds the configuration of the data log. Each record in a data log may hold up to eight fields. The `typesOfFields[]` entry in the structure specifies the types of the fields. Not all the fields are used if fewer than eight elements are declared in this array.

The amount of memory used for a record depends on the number of fields in the record and the size of each field. Use the **datalogRecordSize** function to determine the memory needed for each record.

```
typedef struct datalogConfig_type
{
   UINT16 records;        /* # of records */
   UINT16 fields;         /* # of fields per record */
   DATALOG_VARIABLE typesOfFields[MAX_NUMBER_OF_FIELDS];
}
DATALOG_CONFIGURATION;
```

DATALOG_STATUS

The data log status enumerated type is used to report status information.

```
typedef enum
{
   DLS_CREATED = 0,       /* data log created */
```

```
    DLS_BADID,              /* invalid log ID */
    DLS_EXISTS,             /* log already exists */
    DLS_NOMEMORY,           /* insufficient memory for log */
    DLS_BADCONFIG,          /* invalid configuration */
}
DATALOG_STATUS;
```

DATALOG_VARIABLE

The data log variable enumerated type is used to specify the type of variables to be recorded in the log.

```
typedef enum
{
    DLV_UINT16 = 0,         /* 16 bit unsigned integer */
    DLV_INT16,              /* 16 bit signed integer */
    DLV_UINT32,             /* 32 bit unsigned integer */
    DLV_INT32,              /* 32 bit signed integer */
    DLV_FLOAT,              /* 32 bit floating point */
    DLV_CMITIME,            /* 64 bit time */
    DLV_DOUBLE,             /* 64 bit floating point */
    DLV_NUMBER_OF_TYPES
}
DATALOG_VARIABLE;
```

DialError

The `DialError` enumerated type defines error responses from the dial-up modem functions and may have one of the following values.

```
enum DialError
{
      DE_NoError = 0,
      DE_BadConfig,
      DE_NoModem,
      DE_InitError,
      DE_NoDialTone,
      DE_BusyLine,
      DE_CallAborted,
      DE_FailedToConnect,
      DE_CarrierLost,
      DE_NotInControl
      DE_CallCut
};
```

- `DE_NoError` returns no error has occurred
- `DE_BadConfig` returns the modem configuration structure contains an error
- `DE_NoModem` returns the serial port is not configured as a modem (port type must be RS232_MODEM). Or no modem is connected to the controller serial port.
- `DE_InitError` returns modem initialization failed (the modem may be turned off)
- `DE_NoDialTone` returns modem did not detect a dial tone or the S6 setting in the modem is too short.
- `DE_BusyLine` returns the phone number called was busy
- `DE_CallAborted` returns a call in progress was aborted by the user
- `DE_FailedToConnect` returns the modem could not connect to the remote site

- DE_CarrierLost returns the connection to the remote site was lost (modem reported NO CARRIER). Carrier is lost for a time exceeding the S10 setting in the modem. Phone lines with call waiting are very susceptible to this condition.
- DE_NotInControl returns the serial port is in use by another modem function or has answered an incoming call.
- DE_CallCut returns an incoming call was disconnected while attempting to dial out.

DialState

The DialState enumerated type defines the state of the modemDial operation and may have one of the following values.

```
enum DialState
{
      DS_Inactive,
      DS_Calling,
      DS_Connected,
      DS_Terminating
};
```

- DS_Inactive returns the serial port is not in use by a modem
- DS_Calling returns the controller is making a connection to a remote controller
- DS_Connected returns the controller is connected to a remote controller
- DS_Terminating returns the controller is ending a connection to a remote controller.

DNP_ADDRESS_MAP_TABLE

The dnpAddressMapTable type describes an entry in the DNP Address Mapping Table.

```
typedef struct dnpAddressMapTable_type
{
      UINT16 address;
      CHAR   objectType;
      UINT16 remoteObjectStart;
      UINT16 numberOfPoints;
      UINT16 localModbusAddress;
} dnpAddressMapTable;
```

- address is the DNP station address of the remote station.
- objectType is the DNP object type.
- remoteObjectStart is the DNP address of first object in the remote station.
- numberOfPoints is the number of points.
- localModbusAddress is the Modbus address of first object in local station.

dnpAnalogInput

The dnpAnalogInput type describes a DNP analog input point. This type is used for both 16-bit and 32-bit points.

```
typedef struct dnpAnalogInput_type
{
      UINT16 modbusAddress;
```

```
        UCHAR  class;
        UINT32 deadband;
} dnpAnalogInput;
```

- modbusAddress is the address of the Modbus register number associated with the point.
- class is the reporting class for the object. It may be set to CLASS_1, CLASS_2 or CLASS_3.
- deadband is the amount by which the analog input value must change before an event will be reported for the point.

DnpAnalogInputShortFloat

The dnpAnalogInputShortFloat type describes a DNP analog input point. The format of this point complies with the IEEE-754 standard for floating-point number representation. This type is used for 32-bit points.

```
typedef struct dnpAnalogInputShortFloat_type
{
        UINT16 modbusAddress;
        UCHAR  eventClass;
        float  deadband;
} dnpAnalogInputShortFloat;
```

- modbusAddress is the address of the Modbus register number associated with the point.
- eventClass is the reporting class for the object. It may be set to CLASS_1, CLASS_2 or CLASS_3.
- deadband is the amount by which the analog input value must change before an event will be reported for the point.

dnpAnalogOutput

The dnpAnalogOutput type describes a DNP analog output point. This type is used for both 16-bit and 32-bit points.

```
typedef struct dnpAnalogOutput_type
{
        UINT16 modbusAddress;
} dnpAnalogOutput;
```

- modbusAddress is the address of the Modbus register associated with the point.

dnpBinaryInput

The dnpBinaryInput type describes a DNP binary input point.

```
typedef struct dnpBinaryInput_type
{
        UINT16 modbusAddress;
        UCHAR  class;
} dnpBinaryInput;
```

- modbusAddress is the address of the Modbus register associated with the point.
- class is the reporting class for the object. It may be set to CLASS_1, CLASS_2 or CLASS_3.

dnpBinaryInputEx

The dnpBinaryInputEx type describes an extended DNP Binary Input point.

```
typedef struct dnpBinaryInputEx_type
{
      UINT16 modbusAddress;
      UCHAR  eventClass;
      UCHAR  debounce;
} dnpBinaryInputEx;
```

- modbusAddress is the address of the Modbus register associated with the point.
- class is the reporting class for the object. It may be set to CLASS_1, CLASS_2 or CLASS_3.
- debounceTime is the debounce time for thebinary input.

dnpBinaryOutput

The dnpBinaryOutput type describes a DNP binary output point.

```
typedef struct dnpBinaryOutput_type
{
      UINT16 modbusAddress1;
      UINT16 modbusAddress2;
      UCHAR  controlType;
 } dnpBinaryOutput;
```

- modbusAddress1 is the address of the first Modbus register associated with the point. This field is always used.
- modbusAddress2 is the address of the second Modbus register associated with the point. This field is used only with paired outputs. See the controlType field.
- controlType determines if one or two outputs are associated with this output point. It may be set to PAIRED or NOT_PAIRED.
    - A paired output uses two Modbus registers for output. The first output is the Trip output and the second is the Close output. This is used with Control Relay Output Block objects.
    - A non-paired output uses one Modbus register for output. This is used with Binary Output objects.

dnpConnectionEventType

This enumerated type lists DNP events.

```
typedef enum dnpConnectionEventType
{
      DNP_CONNECTED=0,
      DNP_DISCONNECTED,
      DNP_CONNECTION_REQUIRED,
      DNP_MESSAGE_COMPLETE,
      DNP_MESSAGE_TIMEOUT
} DNP_CONNECTION_EVENT;
```

- The DNP_CONNECTED event indicates that the handler has connected to the master station. The application sends this event to DNP. When DNP receives this event it will send unsolicited messages.
- The DNP_DISCONNECTED event indicates that the handler has disconnected from the master station. The application sends this event to DNP. When DNP receives this event it will request a new connection before sending unsolicited messages.
- The DNP_CONNECTION_REQUIRED event indicates that DNP wishes to connect to the master station. DNP sends this event to the application. The application should process this event by making a connection.
- The DNP_MESSAGE_COMPLETE event indicates that DNP has received confirmation of unsolicited messages from the master station. DNP sends this event to the application. The application should process this event by disconnecting. In many applications a short delay before disconnecting is useful as it allows the master station to send commands to the slave after the unsolicited reporting is complete.
- The DNP_MESSAGE_TIMEOUT event indicates that DNP has attempted to send an unsolicited message but did not receive confirmation after all attempts. This usually means there is a communication problem. DNP sends this event to the application. The application should process this event by disconnecting.

dnpConfiguration

The `dnpConfiguration` type describes the DNP parameters.

```
typedef struct dnpConfiguration_type
{
      UINT16 masterAddress;
      UINT16 rtuAddress;
      CHAR   datalinkConfirm;
      CHAR   datalinkRetries;
      UINT16 datalinkTimeout;
      UINT16 operateTimeout;
      UCHAR  applicationConfirm;
      UINT16 maximumResponse;
      UCHAR  applicationRetries;
      UINT16 applicationTimeout;
      INT16  timeSynchronization;
      UINT16 BI_number;
      UINT16 BI_startAddress;
      CHAR   BI_reportingMethod;
      UINT16 BI_soebufferSize;
      UINT16 BO_number;
      UINT16 BO_startAddress;
      UINT16 CI16_number;
      UINT16 CI16_startAddress;
      CHAR   CI16_reportingMethod;
      UINT16 CI16_bufferSize;
      UINT16 CI32_number;
      UINT16 CI32_startAddress;
      CHAR   CI32_reportingMethod;
      UINT16 CI32_bufferSize;
      CHAR   CI32_wordOrder;
      UINT16 AI16_number;
      UINT16 AI16_startAddress;
      CHAR   AI16_reportingMethod;
```

```
        UINT16 AI16_bufferSize;
        UINT16 AI32_number;
        UINT16 AI32_startAddress;
        CHAR   AI32_reportingMethod;
        UINT16 AI32_bufferSize;
        CHAR   AI32_wordOrder;
        UINT16 AISF_number;
        UINT16 AISF_startAddress;
        CHAR   AISF_reportingMethod;
        UINT16 AISF_bufferSize;
        CHAR   AISF_wordOrder;
        UINT16 AO16_number;
        UINT16 AO16_startAddress;
        UINT16 AO32_number;
        UINT16 AO32_startAddress;
        CHAR   AO32_wordOrder;
        UINT16 AOSF_number;
        UINT16 AOSF_startAddress;
        CHAR   AOSF_wordOrder;
        UINT16 autoUnsolicitedClass1;
        UINT16 holdTimeClass1;
        UINT16 holdCountClass1;
        UINT16 autoUnsolicitedClass2;
        UINT16 holdTimeClass2;
        UINT16 holdCountClass2;
        UINT16 autoUnsolicitedClass3;
        UINT16 holdTimeClass3;
        UINT16 holdCountClass3;
        UINT16 enableUnsolicitedOnStartup;
        UINT16 sendUnsolicitedOnStartup;
        UINT16 level2Compliance;
    } dnpConfiguration;
```

- `masterAddress` is the address of the master station. Unsolicited messages are sent to this station. Solicited messages must come from this station. Valid values are 0 to 65534.
- `rtuAddress` is the address of the RTU. The master station must send messages to this address. Valid values are 0 to 65534.
- `datalinkConfirm` enables requesting data link layer confirmations. Valid values are TRUE and FALSE.
- `datalinkRetries` is the number of times the data link layer will retry a failed message. Valid values are 0 to 255.
- `datalinkTimeout` is the length of time the data link layer will wait for a response before trying again or aborting the transmission. The value is measured in milliseconds. Valid values are 100 to 60000 in multiples of 100 milliseconds.
- `operateTimeout` is the length of time an operate command is valid after receiving a select command. The value is measured in seconds. Valid values are 1 to 6500.
- `applicationConfirm` enables requesting application layer confirmations. Valid values are TRUE and FALSE.
- `maximumResponse` is the maximum length of an application layer response. Valid values are 20 to 2048. The recommended value is 2048 unless the master cannot handle responses this large.

- `applicationRetries` is the number of times the application layer will retry a transmission. Valid values are 0 to 255.
- `applicationTimeout` is the length of time the application layer will wait for a response before trying again or aborting the transmission. The value is measured in milliseconds. Valid values are 100 to 60000 in multiples of 100 milliseconds. This value must be larger than the data link timeout.
- `timeSynchronization` defines how often the RTU will request a time synchronization from the master.
  - Set this to `NO_TIME_SYNC` to disable time synchronization requests.
  - Set this to `STARTUP_TIME_SYNC` to request time synchronization at start up only.
  - Set this to 1 to 32767 to set the time synchronization period in seconds.
- `BI_number` is the number of binary input points. Valid values are 0 to 9999.
- `BI_startAddress` is the DNP address of the first Binary Input point.
- `BI_reportingMethod` determines how binary inputs are reported either Change Of State or Log All Events.
- `BI_soeBufferSize` is the Binary Input Change Event Buffer Size.
- `BO_number` is the number of binary output points. Valid values are 0 to 9999.
- `BO_startAddress` is the DNP address of the first Binary Output point.
- `CI16_number` is the number of 16-bit counter input points. Valid values are 0 to 9999.
- `CI16_startAddress` is the DNP address of the first CI16 point.
- `CI16_reportingMethod` determines how CI16 inputs are reported either Change Of State or Log All Events.
- `CI16_bufferSize` is the number of events in the 16-bit counter change buffer. Valid values are 0 to 9999.
- `CI32_number` is the number of 32-bit counter input points. Valid values are 0 to 9999.
- `CI32_startAddress` is the DNP address of the first CI32 point.
- `CI32_reportingMethod` determines how CI32 inputs are reported either Change Of State or Log All Events.
- `CI32_bufferSize` is the number of events in the 32-bit counter change buffer. Valid values are 0 to 9999.
- `CI32_wordOrder` is the Word Order of CI32 points (0=LSW first, 1=MSW first).
- `AI16_number` is the number of 16-bit analog input points. Valid values are 0 to 9999.
- `AI16_startAddress` is the DNP address of the first AI16 point.
- `AI16_reportingMethod` determines how 16-bit analog changes are reported.

- Set this to `FIRST_VALUE` to report the value of the first change event measured.

- Set this to CURRENT_VALUE to report the value of the latest change event measured.
  - AI16_bufferSize is the number of events in the 16-bit analog input change buffer. Valid values are 0 to 9999.
  - AI32_number is the number of 32-bit analog input points. Valid values are 0 to 9999.
  - AI32_startAddress is the DNP address of the first AI32 point.
  - AI32_reportingMethod determines how 32-bit analog changes are reported.
- Set this to FIRST_VALUE to report the value of the first change event measured.
- Set this to CURRENT_VALUE to report the value of the latest change event measured.
  - AI32_bufferSize is the number of events in the 32-bit analog input change buffer. Valid values are 0 to 9999.
  - AI32_wordOrder is the Word Order of AI32 points (0=LSW first, 1=MSW first)
  - AO16_number is the number of 16-bit analog output points. Valid values are 0 to 9999.
  - AO16_startAddress is the DNP address of the first AO16 point.
  - AO32_number is the number of 32-bit analog output points. Valid values are 0 to 9999.
  - AO32_startAddress is the DNP address of the first AO32 point.
  - AO32_wordOrder is the Word Order of AO32 points (0=LSW first, 1=MSW first)
  - AOSF_number is the number of short float Analog Outputs.
  - AOSF_startAddress is the DNP address of first AOSF point.
  - AOSF_wordOrder is the Word Order of AOSF points (0=LSW first, 1=MSW first).
  - autoUnsolicitedClass1 enables or disables automatic Unsolicited reporting of Class 1 events.
  - holdTimeClass1 is the maximum period to hold Class 1 events before reporting
  - holdCountClass1 is the maximum number of Class 1 events to hold before reporting.
  - autoUnsolicitedClass2 enables or disables automatic Unsolicited reporting of Class 2 events.
  - holdTimeClass2 is the maximum period to hold Class 2 events before reporting
  - holdCountClass2 is the maximum number of Class 2 events to hold before reporting.
  - autoUnsolicitedClass3 enables or disables automatic Unsolicited reporting of Class 3 events.
  - holdTimeClass3 is the maximum period to hold Class 3 events before reporting.
  - holdCountClass2 is the maximum number of Class 3 events to hold before reporting.

- `enableUnsolicitedOnStartup` controls whether unsolicited reporting is initially enabled or disabled in the controller.
- `sendUnsolicitedOnStartup` controls whether a null unsolicited message is sent from the controller on startup.
- `level2Compliance` controls which DNP point types are sent in a Class 0 Poll. If level2Compliance is TRUE, floating point types and 32-bit Analog Outputs are not sent (because they are not level 2 compliant DNP types) – they are converted to 32-bit Analog Inputs and 16-bit Analog Outputs. If level2Compliance is FALSE, all points are reported as their true point type.

dnpConfigurationEx

The `dnpConfigurationEx` type includes extra parameters in the DNP Configuration.

```
typedef struct dnpConfigurationEx_type
{
      UINT16 rtuAddress;
      UCHAR  datalinkConfirm;
      UCHAR  datalinkRetries;
      UINT16 datalinkTimeout;
      UINT16 operateTimeout;
      UCHAR  applicationConfirm;
      UINT16 maximumResponse;
      UCHAR  applicationRetries;
      UINT16 applicationTimeout;
      INT16  timeSynchronization;
      UINT16 BI_number;
      UINT16 BI_startAddress;
      UCHAR  BI_reportingMethod;
      UINT16 BI_soeBufferSize;
      UINT16 BO_number;
      UINT16 BO_startAddress;
      UINT16 CI16_number;
      UINT16 CI16_startAddress;
      UCHAR  CI16_reportingMethod;
      UINT16 CI16_bufferSize;
      UINT16 CI32_number;
      UINT16 CI32_startAddress;
      UCHAR  CI32_reportingMethod;
      UINT16 CI32_bufferSize;
      UCHAR  CI32_wordOrder;
      UINT16 AI16_number;
      UINT16 AI16_startAddress;
      UCHAR  AI16_reportingMethod;
      UINT16 AI16_bufferSize;
      UINT16 AI32_number;
      UINT16 AI32_startAddress;
      UCHAR  AI32_reportingMethod;
      UINT16 AI32_bufferSize;
      UCHAR  AI32_wordOrder;
      UINT16 AISF_number;
      UINT16 AISF_startAddress;
      UCHAR  AISF_reportingMethod;
      UINT16 AISF_bufferSize;
      UCHAR  AISF_wordOrder;
```

```
    UINT16 AO16_number;
    UINT16 AO16_startAddress;
    UINT16 AO32_number;
    UINT16 AO32_startAddress;
    UCHAR  AO32_wordOrder;
    UINT16 AOSF_number;
    UINT16 AOSF_startAddress;
    UCHAR  AOSF_wordOrder;
    UINT16 autoUnsolicitedClass1;
    UINT16 holdTimeClass1;
    UINT16 holdCountClass1;
    UINT16 autoUnsolicitedClass2;
    UINT16 holdTimeClass2;
    UINT16 holdCountClass2;
    UINT16 autoUnsolicitedClass3;
    UINT16 holdTimeClass3;
    UINT16 holdCountClass3;
    UINT16 enableUnsolicitedOnStartup;
    UINT16 sendUnsolicitedOnStartup;
    UINT16 level2Compliance;
    UINT16 masterAddressCount;
    UINT16 masterAddress[8];
    UINT16 maxEventsInResponse;
    UINT16 dialAttempts;
    UINT16 dialTimeout;
    UINT16 pauseTime;
    UINT16 onlineInactivity;
    UINT16 dialType;
    Char   modemInitString[64];
} dnpConfigurationEx;
```

- `rtuAddress` is the address of the RTU. The master station must send messages to this address. Valid values are 0 to 65534.
- `datalinkConfirm` enables requesting data link layer confirmations. Valid values are TRUE and FALSE.
- `datalinkRetries` is the number of times the data link layer will retry a failed message. Valid values are 0 to 255.
- `datalinkTimeout` is the length of time the data link layer will wait for a response before trying again or aborting the transmission. The value is measured in milliseconds. Valid values are 100 to 60000 in multiples of 100 milliseconds.
- `operateTimeout` is the length of time an operate command is valid after receiving a select command. The value is measured in seconds. Valid values are 1 to 6500.
- `applicationConfirm` enables requesting application layer confirmations. Valid values are TRUE and FALSE.
- `maximumResponse` is the maximum length of an application layer response. Valid values are 20 to 2048. The recommended value is 2048 unless the master cannot handle responses this large.
- `applicationRetries` is the number of times the application layer will retry a transmission. Valid values are 0 to 255.
- `applicationTimeout` is the length of time the application layer will wait for a response before trying again or aborting the transmission. The value is measured in

milliseconds. Valid values are 100 to 60000 in multiples of 100 milliseconds. This value must be larger than the data link timeout.

- `timeSynchronization` defines how often the RTU will request a time synchronization from the master.
  - Set this to `NO_TIME_SYNC` to disable time synchronization requests.
  - Set this to `STARTUP_TIME_SYNC` to request time synchronization at start up only.
  - Set this to 1 to 32767 to set the time synchronization period in seconds.
- `BI_number` is the number of binary input points. Valid values are 0 to 9999.
- `BI_startAddress` is the DNP address of the first Binary Input point.
- `BI_reportingMethod` determines how binary inputs are reported either Change Of State or Log All Events.
- `BI_soebufferSize` is the Binary Input Change Event Buffer Size.
- `BO_number` is the number of binary output points. Valid values are 0 to 9999.
- `BO_startAddress` is the DNP address of the first Binary Output point.
- `CI16_number` is the number of 16-bit counter input points. Valid values are 0 to 9999.
- `CI16_startAddress` is the DNP address of the first CI16 point.
- `CI16_reportingMethod` determines how CI16 inputs are reported either Change Of State or Log All Events.
- `CI16_bufferSize` is the number of events in the 16-bit counter change buffer. Valid values are 0 to 9999.
- `CI32_number` is the number of 32-bit counter input points. Valid values are 0 to 9999.
- `CI32_startAddress` is the DNP address of the first CI32 point.
- `CI32_reportingMethod` determines how CI32 inputs are reported either Change Of State or Log All Events.
- `CI32_bufferSize` is the number of events in the 32-bit counter change buffer. Valid values are 0 to 9999.
- `CI32_wordOrder` is the Word Order of CI32 points (0=LSW first, 1=MSW first).
- `AI16_number` is the number of 16-bit analog input points. Valid values are 0 to 9999.
- `AI16_startAddress` is the DNP address of the first AI16 point.
- `AI16_reportingMethod` determines how 16-bit analog changes are reported.

- Set this to `FIRST_VALUE` to report the value of the first change event measured.
- Set this to `CURRENT_VALUE` to report the value of the latest change event measured.
  - `AI16_bufferSize` is the number of events in the 16-bit analog input change buffer. Valid values are 0 to 9999.

- AI32_number is the number of 32-bit analog input points. Valid values are 0 to 9999.
- AI32_startAddress is the DNP address of the first AI32 point.
- AI32_reportingMethod determines how 32-bit analog changes are reported.

- Set this to FIRST_VALUE to report the value of the first change event measured.
- Set this to CURRENT_VALUE to report the value of the latest change event measured.
  - AI32_bufferSize is the number of events in the 32-bit analog input change buffer. Valid values are 0 to 9999.
  - AI32_wordOrder is the Word Order of AI32 points (0=LSW first, 1=MSW first)
  - AISF_number is the number of short float Analog Inputs.
  - AISF_startAddress is the DNP address of first AISF point.
  - AISF_reportingMethod is the event reporting method, Change Of State or Log All Events.
  - AISF_bufferSize is the short float Analog Input Event Buffer Size.
  - AISF_wordOrder is the word order of AISF points (0=LSW first, 1=MSW first) */
  - AO16_number is the number of 16-bit analog output points. Valid values are 0 to 9999.
  - AO16_startAddress is the DNP address of the first AO16 point.
  - AO32_number is the number of 32-bit analog output points. Valid values are 0 to 9999.
  - AO32_startAddress is the DNP address of the first AO32 point.
  - AO32_wordOrder is the Word Order of AO32 points (0=LSW first, 1=MSW first)
  - AOSF_number is the number of short float Analog Outputs.
  - AOSF_startAddress is the DNP address of first AOSF point.
  - AOSF_wordOrder is the Word Order of AOSF points (0=LSW first, 1=MSW first).
  - autoUnsolicitedClass1 enables or disables automatic Unsolicited reporting of Class 1 events.
  - holdTimeClass1 is the maximum period to hold Class 1 events before reporting
  - holdCountClass1 is the maximum number of Class 1 events to hold before reporting.
  - autoUnsolicitedClass2 enables or disables automatic Unsolicited reporting of Class 2 events.
  - holdTimeClass2 is the maximum period to hold Class 2 events before reporting
  - holdCountClass2 is the maximum number of Class 2 events to hold before reporting.
  - autoUnsolicitedClass3 enables or disables automatic Unsolicited reporting of Class 3 events.

- holdTimeClass3 is the maximum period to hold Class 3 events before reporting.
- HoldCountClass3 is the maximum number of Class 3 events to hold before reporting.
- EnableUnsolicitedOnStartup enables or disables unsolicited reporting at start-up.
- SendUnsolicitedOnStartup sends an unsolicited report at start-up.
- level2Compliance reports only level 2 compliant data types (excludes floats, AO-32).
- MasterAddressCount is the number of master stations.
- masterAddress[8] is the number of master station addresses.
- MaxEventsInResponse is the maximum number of change events to include in read response.
- PSTNDialAttempts is the maximum number of dial attempts to establish a PSTN connection.
- PSTNDialTimeout is the maximum time after initiating a PSTN dial sequence to wait for a carrier signal.
- PSTNPauseTime is the pause time between dial events.
- PSTNOnlineInactivity is the maximum time after message activity to leave a PSTN connection open before hanging up.
- PSTNDialType is the dial type: tone or pulse dialling.
- modemInitString[64] is the initialization string to send to the modem.

dnpCounterInput

The dnpCounterInput type describes a DNP counter input point. This type is used for both 16-bit and 32-bit points.

```
typedef struct dnpCounterInput_type
{
      UINT16 modbusAddress;
      UCHAR  class;
      UINT32 threshold;
 } dnpCounterInput;
```

- modbusAddress is the address of the Modbus register number associated with the point.
- class is the reporting class for the object. It may be set to CLASS_1, CLASS_2 or CLASS_3.
- threshold is the amount by which the counter input value must change before an event will be reported for the point.

dnpMasterPoll

The dnpMasterPoll type describes an entry in the DNP Master Poll Table.

```
typedef struct dnpMasterPoll_type
{
      UINT16 dnpRemoteStationAddress;
```

```
      UINT16 class0PollRate;
      UINT16 class1PollRate;
      UINT16 class2PollRate;
      UINT16 class3PollRate;
      UINT16 timeSyncRate;
      UINT16 unsolicitedResponseFlags;
} dnpMasterPoll;
```

- `dnpRemoteStationAddress` is the remote DNP station address.
- `class0PollRate` is the Class 0 Polling rate.
- `class1PollRate` is the Class 1 Polling rate.
- `class2PollRate` is the Class 2 Polling rate.
- `class3PollRate` is the Class 3 Polling rate.
- `timeSyncRate` is the time synchronization rate.
- `unsolicitedResponseFlags` are the DNP Master Unsolicited Response enable flags.

DNP Master Poll table Extended Entry

The `dnpMasterPollEx` type describes an extended entry in the DNP Master Poll Table.

```
typedef struct dnpMasterPollTableEx_type
{
      INT16 dnpRemoteStationAddress;
      INT16 class0PollRate;
      INT16 class1PollRate;
      INT16 class2PollRate;
      INT16 class3PollRate;
      INT16 timeSyncRate;
      UINT16 unsolicitedResponseFlags;
      UINT16 class0PollOffset;
      UINT16 class1PollOffset;
      UINT16 class2PollOffset;
      UINT16 class3PollOffset;
      UINT16 timeSyncOffset;
      INT16 class1MaxEvents;
      INT16 class2MaxEvents;
      INT16 class3MaxEvents;
      UINT16 saveIINFlagsRegister;
} dnpMasterPollTableEx;
```

- `dnpRemoteStationAddress` is the remote DNP station address.
- `class0PollRate` is the Class 0 Polling rate.
- `class1PollRate` is the Class 1 Polling rate.
- `class2PollRate` is the Class 2 Polling rate.
- `class3PollRate` is the Class 3 Polling rate.
- `timeSyncRate` is the time synchronization rate.

- `unsolicitedResponseFlags` are the DNP Master Unsolicited Response enable flags.
- `TimeSyncRate is the time synchronisation rate.`
- `unsolicitedResponseFlags are the flags for enabling Unsolicited Responses.`
- `class0PollOffset` is the offset for Class 0 Polling.
- `class1PollOffset` is the offset for Class 1 Polling.
- `class2PollOffset` is the offset for Class 2 Polling.
- `class3PollOffset` is the offset for Class 3 Polling.
- `timeSyncOffset` is the offset for time synchronization.
- `class1MaxEvents` is the maximum limit of Class 1 events in poll response.
- `class2MaxEvents` is the maximum limit of Class 2 events in poll response.
- `class3MaxEvents` is the maximum limit of Class 3 events in poll response.
- `saveIINFlagsRegister.`

dnpPointType

The enumerated type `DNP_POINT_TYPE` includes all allowed DNP data point types.

```
typedef enum dnpPointType
{
      BI_POINT=0,       /* binary input */
      AI16_POINT,       /* 16 bit analog input */
      AI32_POINT,       /* 32 bit analog input */
      AISF_POINT,       /* short float analog input */
      AILF_POINT,       /* long float analog input */
      CI16_POINT,       /* 16 bit counter output */
      CI32_POINT,       /* 32 bit counter output */
      BO_POINT,         /* binary output */
      AO16_POINT,       /* 16 bit analog output */
      AO32_POINT,       /* 32 bit analog output */
      AOSF_POINT,       /* short float analog output */
      AOLF_POINT        /* long float analog output */
} DNP_POINT_TYPE;
```

dnpProtocolStatus

The dnpPrototocolStatus structure contains status information for DNP message transactions.

```
struct dnpPrototocolStatus {
      UINT16 successes;
      UINT16 failures;
      UINT16 failuresSinceLastSuccess;
      UINT16 formatErrors;
      UINT16 framesReceived;
      UINT16 framesSent;
      UINT16 messagesReceived;
```

```
        UINT16 messagesSent;
};
```

- successes is the number of successful DNP message transactions
- failures is the total number of failed DNP message transactions
- failuresSinceLastSuccess is the number of failures since last the success
    - formatErrors is the number of messages received with bad message data.
    - framesReceived is the number of DNP frames (message packets) received.
    - framesSent is the number of DNP frames (message packets) sent.
    - messagesReceived is the number of DNP messages received.
    - messagesSent is the number of DNP messages sent.
    - commandStatus is the status of the last protocol command sent.

dnpRoutingTableEx

The dnpRoutingTableEx type describes an entry in the DNP Routing Table. The DNP Routing Table is a list of routes, which are maintained in ascending order of DNP addresses.

```
typedef struct RoutingTableEx_type
{
      UINT16 address;         // station address
      UINT16 comPort;         // com port interface
      UINT16 retries;         // number of retries
      UINT16 timeout;         // timeout in milliseconds
      IP_ADDRESS ipAddress;   // IP address
} dnpRoutingTableEx;
```

- address is the DNP station address of the destination station.
- comPort specifies the communications port interface. Allowed values are :
  1 = serial port com1
  2 = serial port com2
  3 = serial port com3
  103 = DNP over TCP, using LAN port
  104 = DNP over UDP, using LAN port
- retries is the number of times the data link layer will retry the message in the event of a failure.
- timeout is the timeout in milliseconds.
- ipAddress is the IP address of the destination station.

DNP_RUNTIME_STATUS

The dnpRuntimeStatus type describes a structure for holding status information about DNP event log buffers.

```
/* DNP Runtime Status */
typedef struct dnp_runtime_status
{
```

```
    UINT16 eventCountBI;     /* number of binary input events */
    UINT16 eventCountCI16;   /* number of 16-bit counter events */
    UINT16 eventCountCI32;   /* number of 32-bit counter events */
    UINT16 eventCountAI16;   /* number of 16-bit analog input
                                events */
    UINT16 eventCountAI32;   /* number of 32-bit analog input
                                events */
    UINT16 eventCountAISF;   /* number of short floating-point
                                analog input events */
    UINT16 eventCountClass1; /* number of class 1 events */
    UINT16 eventCountClass2; /* number of class 2 events */
    UINT16 eventCountClass3; /* number of class 3 events */
} DNP_RUNTIME_STATUS;
```

- `eventCountBI` is number of binary input events.
- `eventCountCI16` is number of 16-bit counter events.
- `eventCountCI32` is number of 32-bit counter events.
- `eventCountAI16` is number of 16-bit analog input events.
- `eventCountAI32` is number of 32-bit analog input events.
- `EventCountAISF` is number of short floating-point analog input events.
- `eventCountClass1` is the class 1 event counter.
- `eventCountClass2` is the class 2 event counter.
- `eventCountClass3` is the class 3 event counter.

envelope

The envelope type is a structure containing a message envelope. Envelopes are used for inter-task communication.

```
typedef struct envelope_type {
      UINT32 source;         // sender task ID
      UINT32 destination;    // destination task ID
      UINT32 type;           / type of message
      UINT32 data;           // the message data
      }
      envelope;
```

- `link` is a pointer to the next envelope in a queue. This field is used by the RTOS. It is of no interest to an application program.
- `source` is the task ID of the task sending the message. This field is specified automatically by the `send_message` function. The receiving task may read this field to determine the source of the message.
- `destination` is the task ID of the task to receive the message. It must be specified before calling the `send_message` function.
- `type` specifies the type of data in the data field. It may be MSG_DATA, MSG_POINTER, or any other value defined by the application program. This field is not required.
- `data` is the message data. The field may contain a datum or pointer. The application program determines the use of this field.

HART_COMMAND

The HART_COMMAND type is a structure containing a command to be sent to a HART slave device. The command field contains the HART command number. The length field contains the length of the data string to be transmitted (the byte count in HART documentation). The data field contains the data to be sent to the slave.

```
typedef struct hartCommand_t
      {
      UINT16 command;
      UINT16 length;
      CHAR   data[DATA_SIZE];
      }
      HART_COMMAND;
```

- `command` is the HART command number.
- `length` is the number of characters in the data string.
- `data[DATA_SIZE]` is the data field for the command.

HART_DEVICE

The HART_DEVICE type is a structure containing information about the HART device. The information is read from the device using command 0 or command 11. The fields are identical to those read by the commands. Refer to the command documentation for more information.

```
typedef struct hartDevice_t
      {
      UCHAR manufacturerID;
      UCHAR manufacturerDeviceType;
      UCHAR preamblesRequested;
      UCHAR commandRevision;
      UCHAR transmitterRevision;
      UCHAR softwareRevision;
      UCHAR hardwareRevision;
      UCHAR flags;
      UINT32 deviceID;
      }
      HART_DEVICE;
```

HART_RESPONSE

The HART_RESPONSE type is a structure containing a response from a HART slave device. The command field contains the HART command number. The length field contains the length of the data string to be transmitted (the byte count in HART documentation). The data field contains the data to be sent to the slave.

```
typedef struct hartResponse_t
      {
      UINT16 code;
      UINT16 length;
      CHAR *   pData;
      }
      HART_RESPONSE;
```

- `response` is the response code from the device.
- `length` is the length of response data.

- `data[DATA_SIZE]` is the data field for the response.

HART_RESULT

The HART_RESULT enumeration type defines a list of results of sending a command.

```
typedef enum hartResult_t
      {
      HR_NoModuleResponse=0,
      HR_CommandPending,
      HR_CommandSent,
      HR_Response,
      HR_NoResponse,
      HR_WaitTransmit
      }
      HART_RESULT;
```

- `HR_NoModuleResponse` returns no response from HART modem module.
- `HR_CommandPending` returns command ready to be sent, but not sent.
- `HR_CommandSent` returns command sent.
- `HR_Response` returns response received.
- `HR_NoResponse` returns no response after all attempts.
- `HR_WaitTransmit` returns modem is not ready to transmit.

HART_SETTINGS

The HART_SETTINGS type is a structure containing the configuration for the HART modem module. The `useAutoPreamble` field indicates if the number of preambles is set by the value in the HART_SETTINGS structure (FALSE) or the value in the HART_DEVICE structure (TRUE). The `deviceType` field determines if the 5904 modem is a HART primary master or secondary master device (primary master is the recommended setting).

```
typedef struct hartSettings_t
      {
      UINT16 attempts;
      UINT16 preambles;
      BOOLEAN  useAutoPreamble;
      UINT16 deviceType;
      }
      HART_SETTINGS;
```

- `attempts` is the number of command attempts (1 to 4).
- `preambles` is the number of preambles to send (2 to 15).
- `useAutoPreamble` is a flag to use the requested preambles.
- `deviceType` is the type of HART master (1 = primary; 0 = secondary).

HART_VARIABLE

The HART_VARIABLE type is a structure containing a variable read from a HART device. The structure contains three fields that are used by various commands. Note that not all fields will be used by all commands. Refer to the command specific documentation.

```
typedef struct hartVariable_t
      {
      float    value;
      UINT16 units;
      UINT16 variableCode;
      }
      HART_VARIABLE;
```

- `value` is the value of the variable.
- `units` are the units of measurement.
- `variableCode` is the transmitter specific variable ID.

IO_CONFIG Structure

The `IO_CONFIG` structure contains I/O System configuration data.

```
typedef struct{
      UINT16 slaveAddress;
      UINT16 dataRate;
      UINT16 numberOfAttempts;
      UINT16 ledPower;
}IO_CONFIG;
```

- `slaveAddress` returns the I^2C address, 0 = slave mode disabled
- `dataRate` returns the I/O bus data rate 0 = 100 kHz ;1 = 150 kHz; 2 = 200 kHz; 3 = 250 kHz; 4 = 300 kHz; 5 = 350 kHz; 6 = 400 kHz (default); 7 = 450 kHz;
- `numberOfAttempts` returns the number of attempts, 1 to 4 (default = 1)
- `ledPower` returns the led power state, 0 = off, 1 = on (default)

IO_STATUS Structure

The `IO_STATUS` structure contains status information from the last scan of a specific I/O module.

```
typedef struct{
      UINT16 commStatus;
      UINT32 scanTime;
}IO_STATUS;
```

The IO_STATUS structure contains the following data fields.

- `commStatus` returns the communication status, 0=failed, 1=success
- `scanTime` returns time of last scan in milliseconds according to the stop watch clock

IP_ADDRESS

The IP Address structure defines an IPv4 address. This is the standard IPv4 address structure used by sockets APIs and is also used by Modbus/TCP C++ Tools functions .

```
struct in_addr
{
    u_long s_addr;
};
typedef struct in_addr     IP_ADDRESS;
```

- `s_addr` is a 32bit netis/hostid address in network byte order.

IP_CONNECTION_SUMMARY

The IP Connection Summary structure summarizes the number and type of active TCP/IP connections.

```
typedef struct st_connectionSummary
   {
   UINT32 slaveConnections;
   UINT32 masterConnections;
   UINT32 unusedConnections;
   }
   IP_CONNECTION_SUMMARY;
```

- `slaveConnections` is the number of active slave TCP/IP connections.
- `masterConnections` is the number of active master TCP/IP connections.
- `unusedConnections` is the number of unused TCP/IP connections available.

IP_CONFIG_MODE Enumeration

The IP_CONFIG_MODE enumeration defines IP configuration options.

```
typedef enum ipConfigMode_t
{
      IPConfig_CtrlSettings  = 0,
      IPConfig_GatewayOnLAN  = 0,
      IPConfig_GatewayOnCom1 = 1,
      IPConfig_GatewayOnCom2 = 2,
      IPConfig_GatewayOnCom3 = 3,
      IPConfig_GatewayOnCom4 = 4
}
IP_CONFIG_MODE;
```

- IPConfig_CtrlSettings configures IP settings from controller settings. Default gateway is on LAN subnet. IP_SETTINGS defines gateway address. Same as IPConfig_GatewayOnLAN.
- IPConfig_GatewayOnLAN configures IP settings from controller settings. Default gateway is on LAN subnet. IP_SETTINGS defines gateway address. Same as IPConfig_CtrlSettings.
- IPConfig_GatewayOnCom1 configures IP settings from controller settings. Default gateway is the com1 PPP connection.
- IPConfig_GatewayOnCom2 configures IP settings from controller settings. Default gateway is the com2 PPP connection.
- IPConfig_GatewayOnCom3 configures IP settings from controller settings. Default gateway is the com3 PPP connection.
- IPConfig_GatewayOnCom4 configures IP settings from controller settings. Default gateway is the com4 PPP connection.

IP_PROTOCOL_SETTINGS

The Modbus IP Protocol Settings structure defines settings for one of the Modbus IP communication protocols.

```
typedef struct st_ipProtocolSettings
{
      UINT16    portNumber;
      UINT32    masterIdleTimeout;
      UINT32    serverIdleTimeout;
      BOOLEAN   serverEnabled;
}
IP_PROTOCOL_SETTINGS;
```

- `portNumber` is the TCP or UDP port number for the Modbus IP of DNP IP protocol. Valid port numbers are 1 to 65535.
- `masterIdleTimeout` is the length of time, in seconds, that a master connection will wait for the user to send the next command before ending the connection. This allows the slave device to free unused connections while the master application may retain the connection allocation. Set to 0 to disable timeout and let the application close the connection. Valid values are any 32-bit integer. Default value is 10 seconds. TCP protocols only. Not used by UDP protocols.
- `serverIdleTimeout` is the length of time, in seconds, that a server connection will wait for a message before ending the connection. Set to 0 to disable timeout and let remote client close connection. Valid values are any 32-bit integer. Default value is 250 seconds. TCP protocols only. Not used by UDP protocols.
- `serverEnabled` is the enable server control flag.

IP_PROTOCOL_TYPE

The IP_PROTOCOL_TYPE enumerated type defines TCP/IP protocols supported by the SCADAPack2.

```
typedef enum ipProtocol_t
{
      PP_None = 0,
      IPP_ModbusTcp,
      IPP_ModbusRtuOverUdp,
      IPP_ModbusAsciiOverUdp,
      IPP_DnpOverTcp,
      IPP_DnpOverUdp
}
IP_PROTOCOL_TYPE;
```

IP_SETTINGS

The IP Settings structure defines IP settings for a communication interface installed on the TCP/IP stack.

```
typedef struct st_IPSettings
{
      IP_CONFIG_MODE   ipConfigMode;
      UINT32           ipAddress[4];
      UINT32           gateway[4];
      UINT32           netMask;
      UCHAR            ipVersion;
}
IP_SETTINGS;
```

- ipConfigMode are the IP configuration options. See the IP_CONFIG_MODE enumeration for values supported.
- ipAddress is the IP address. Only the first 32-bit value in this array is supported and contains IP address in form of 32-bit unsigned integer. For example IP address 172.016.017.018 will be represented with following 32-bit unsigned number:

  172 + **16**x256 + **17**x256x256 + **18**x256x256x256 = 303108268
- gateway is the network gateway. Only the first 32-bits are supported.
- netMask is the subnet mask.
- ipVersion is the IP version. Only the value 4 is supported for IP version 4.

ledControl_tag

The `ledControl_tag` structure defines LED power control parameters.

```
struct ledControl_tag
{
      UINT16 state;
      UINT16 time;
};
```

- `state` is the default LED state. It is either the LED_ON or LED_OFF macro.
- `time` is the period, in minutes, after which the LED power returns to its default state.

MASTER_MESSAGE

The MASTER_MESSAGE structure defines a Modbus serial master message.

```
typedef struct st_masterMessage
{
      FILE *      stream;           // serial port
      UINT16      function;         // Modbus function code
      UINT16      slaveStation;     // slave station address
      UINT16      slaveRegister;    // slave Modbus register
      UINT16      masterRegister;   // master Modbus register
      UINT16      length;           // number of registers
      UINT16      timeout;          // time to wait for response
      in tenths of seconds
      BOOLEAN     eventRequest;     // signal event on completion
      (optional)
      UINT32      eventNo;          // event to signal when
      timeout or response received (optional)
}
MASTER_MESSAGE;
```

- *stream* is the serial port to send the command message. Valid values are: com1, com2, and com3.

- *function* specifies the Modbus function code. Refer to the communication protocol manual for supported function codes.
- *slaveStation* specifies the address of the slave station.
- *slaveRegister* specifies the location of data in the slave station. Depending on the Modbus function code, data may be read or written at this location.

- *masterRegister* specifies the location of data in the master (this controller). Depending on the function code, data may be read or written at this location.
- *length* specifies the number of registers.
- *timeout* specifies how long in tenths of seconds to wait for a response.
- *eventRequest* requests an event to be signaled on completion. If set to TRUE, the *eventNo* will be signaled when the response is received or a timeout has occurred. Set to FALSE to disable this feature.
- *eventNo* specifies the event to signal on completion. This field is only used if *eventRequest* is set to TRUE.

MODBUS_CMD_STATUS

The master command status codes have been changed from macros to the enumeration type MODBUS_CMD_STATUS. The previously supported status codes have the same value as they did as a macro.

```
typedef enum modbusCmdStatus_t
{
   MM_SENT                   = 0,
   MM_RECEIVED               = 1,
   MM_NO_MESSAGE             = 2,
   MM_BAD_FUNCTION           = 3,
   MM_BAD_SLAVE              = 4,
   MM_BAD_ADDRESS            = 5,
   MM_BAD_LENGTH             = 6,
   MM_PROTOCOL_NOT_SUPPORTED = 7,

// additional master command status codes used for Modbus/TCP
master messaging only
   MM_CONNECTING             = 8,
   MM_CONNECTED              = 9,
   MM_CONNECT_TIMEOUT        = 10,
   MM_SEND_ERROR             = 11,
   MM_RSP_TIMEOUT            = 12,
   MM_RSP_ERROR              = 13,
   MM_DISCONNECTING          = 14,
   MM_DISCONNECTED           = 15,
   MM_BAD_CONNECT_ID         = 16,
   MM_BAD_PROTOCOL_TYPE      = 17,
   MM_BAD_IP_ADDRESS         = 18,
   MM_BUSY                   = 19,
   MM_ENDED                  = 20,
   MM_CONNECT_ERROR          = 21,
   MM_NO_MORE_CONNECTIONS    = 22,
   MM_BAD_CONNECTION_TYPE    = 23,
   MM_EXCEPTION_FUNCTION     = 24,
   MM_EXCEPTION_ADDRESS      = 25,
   MM_EXCEPTION_VALUE        = 26,
   MM_QUEUE_FULL             = 27,
   MM_STATIONS_ARE_EQUAL     = 28
}
MODBUS_CMD_STATUS;
```

- `MM_SENT` returns a valid command has been sent
- `MM_RECEIVED` returns response was received.
- `MM_NO_MESSAGE` returns no message was sent.

- MM_BAD_FUNCTION returns invalid function code used
- MM_BAD_SLAVE returns invalid slave station address used
- MM_BAD_ADDRESS returns invalid database address used
- MM_BAD_LENGTH returns invalid message length
- MM_PROTOCOL_NOT_SUPPORTED returns selected protocol is not supported.
- MM_CONNECTING returns connecting to slave IP address.
- MM_CONNECTED returns connected to slave IP address.
- MM_CONNECT_TIMEOUT returns timeout while connecting to slave IP address.
- MM_SEND_ERROR returns TCP/IP error has occurred while sending message.
- MM_RSP_TIMEOUT returns timeout has occurred waiting for response.
- MM_RSP_ERROR returns slave has closed connection; incorrect response; or, incorrect response length.
- MM_DISCONNECTING returns disconnecting from slave IP address is in progress.
- MM_DISCONNECTED returns connection to slave IP address is disconnected.
- MM_BAD_CONNECT_ID returns invalid connection ID.
- MM_BAD_PROTOCOL_TYPE returns invalid protocol type.
- MM_BAD_IP_ADDRESS returns invalid slave IP address.
- MM_BUSY returns last message is still being processed.
- MM_ENDED returns Master connection has been released. This status is only reported by the ISaGRAF masterIP function block. It is not available from the mTcpMasterStatus function.
- MM_CONNECT_ERROR returns error while connecting to slave IP address.
- MM_NO_MORE_CONNECTIONS returns no more connections are available.
- MM_BAD_CONNECTION_TYPE returns invalid connection type used in mTcpMasterMessage.
- MM_EXCEPTION_FUNCTION Returns master message status: Modbus slave returned a function exception
- MM_EXCEPTION_ADDRESS Returns master message status: Modbus slave returned an address exception
- MM_EXCEPTION_VALUE Returns master message status: Modbus slave returned a value exception
- MM_QUEUE_FULL Returns master message status: Serial transmit queue is full
- MM_STATIONS_ARE_EQUAL Returns master message status: Master and slave stations are equal. They must be different.

ModemInit

The ModemInit structure specifies modem initialization parameters for the modemInit function.

```
struct ModemInit
{
```

```
        FILE * port;
        CHAR   modemCommand[MODEM_CMD_MAX_LEN + 2];
};
```

- `port` is the serial port where the modem is connected.
- `modemCommand` is the initialization string for the modem. The characters AT will be prefixed to the command, and a carriage returned suffixed to the command when it is sent to the modem. Refer to the section **Modem Commands** for suggested command strings for your modem.

ModemSetup

The `ModemSetup` structure specifies modem initialization and dialing control parameters for the `modemDial` function.

```
struct ModemSetup
{
        FILE *    port;
        UINT16 dialAttempts;
        UINT16 detectTime;
        UINT16 pauseTime;
        UINT16 dialmethod;
        CHAR      modemCommand[MODEM_CMD_MAX_LEN + 2];
        CHAR      phoneNumber[PHONE_NUM_MAX_LEN + 2];
};
```

- `port` is the serial port where the modem is connected.
- `dialAttempts` is the number of times the controller will attempt to dial the remote controller before giving up and reporting an error.
- `detectTime` is the length of time in seconds that the controller will wait for carrier to be detected. It is measured from the start of the dialing attempt.
- `pauseTime` is the length of time in seconds that the controller will wait between dialing attempts.
- `dialmethod` selects pulse or tone dialing. Set `dialmethod` to 0 for tone dialing or 1 for pulse dialing.
- `modemCommand` is the initialization string for the modem. The characters AT will be prepended to the command, and a carriage returned appended to the command when it is sent to the modem. Refer to the section **Modem Commands** for suggested command strings for your modem.
- `phoneNumber` is the phone number of the remote controller. The characters ATD and the dialing method will be prepended to the command, and a carriage returned appended to the command when it is sent to the modem.

MTCP_CONFIGURATION

The Modbus/TCP Settings structure defines settings for the Modbus/TCP communication protocol.

```
typedef struct st_ModbusTcpSettings
   {
   UINT16    portNumber;
   UINT32    masterIdleTimeout;
   UINT32    slaveRecvTimeout;
```

```
        UINT32    maxServerConnections;
        }
        MTCP_CONFIGURATION;
```

- `portNumber` is the Modbus/TCP protocol port number. Valid port numbers are 0 to 65535. Selecting port number 65535 allows a server to listen for incoming connection requests on all the ports. Default port number is 502.
- `masterIdleTimeout` is the length of time, in seconds, that a master connection will wait for the user to send the next command before ending the connection. Set to 0 to disable timeout and let application close the connection. Valid values are any 32-bit integer. Default value is 10 seconds.
- `slaveRecvTimeout` is the length of time, in seconds, that a server connection will wait for a message before ending the connection. Set to 0 to disable timeout and let remote client close connection. Valid values are any 32-bit integer. Default value is 10 seconds.

`maxServerConnections` is the maximum number of connections allowed by the server at once. Default value is 20.

MTCP_IF_SETTINGS

The Modbus IP Interface Settings structure defines the interface settings when using any Modbus IP protocol on the specified interface.

```
typedef struct st_MTcpIfSettings
{
      UINT16    station;
      UCHAR     addrMode;
      BOOLEAN   sfMessaging;
}
MTCP_IF_SETTINGS;
```

- `station` is the Modbus station address for the specified communication interface. Valid values are 1 to 255 in standard Modbus, 1 to 65534 in extended Modbus. Default value is 1.
- `addrMode` is the addressing mode, AM_standard or AM_extended. Default value is AM_standard.
- SF`Messaging` is the enable Store and Forward messaging control flag. Enable store and forward when set to TRUE. Disable store and forward when set to FALSE. Default value is FALSE.

MTCP_IF_SETTINGS_EX

The Modbus IP Interface Extended Settings structure defines the interface settings when using any Modbus IP protocol on the specified interface. This structure includes Enron Modbus support.

```
typedef struct st_MTcpIfSettingsEx_type
{
      UINT16    station;
      UCHAR     addrMode;
      BOOLEAN   sfMessaging;
      BOOLEAN   enronEnabled;
      UINT16    enronStation;
}
MTCP_IF_SETTINGS_EX;
```

- `station` is the Modbus station address for the specified communication interface. Valid values are 1 to 255 in standard Modbus, 1 to 65534 in extended Modbus. Default value is 1.
- `addrMode` is the addressing mode, AM_standard or AM_extended. Default value is AM_standard.
- `SFMessaging` is the enable Store and Forward messaging control flag. Enable store and forward when set to TRUE. Disable store and forward when set to FALSE. Default value is FALSE.
- `enronEnabled` determines if the Enron Modbus station is enabled. It may be `TRUE` or FALSE.
- `enronStation` is the station address for the Enron Modbus protocol. It is used if `enronEnabled` is set to `TRUE`. Valid values are 1 to 255 for standard addressing, and 1 to 65534 for extended addressing.

pconfig

The `pconfig` structure contains serial port settings.

```
struct pconfig {
    UINT16 baud;
    UINT16 duplex;
    UINT16 parity;
    UINT16 data_bits;
    UINT16 stop_bits;
    UINT16 flow_rx;
    UINT16 flow_tx;
    UINT16 type;
    UINT16 timeout;
    };
```

- `baud` is the communication speed. It is one of the `BAUDxxx` macros.
- `duplex` is either the FULL or HALF macro.
- `parity` is one of NONE, EVEN or ODD macros.
- `data_bits` is the word length. It is either the DATA7 or DATA8 macro.
- `stop_bits` in the number of stop bits transmitted. The only supported selection is the STOP1 macro.
- `flow_rx` specifies flow control on the receiver. It is either the RFC_MODBUS_RTU (=ENABLE), or RFC_NONE (=DISABLE). If the Modbus RTU protocol is used, set `flow_rx` to RFC_MODBUS_RTU. For the Modbus ASCII protocol or any other protocol, set `flow_rx` to RFC_NONE.
- `flow_tx` specifies flow control on the transmitter. It is either the TFC_IGNORE_CTS (=ENABLE) or TFC_NONE (=DISABLE) macro. Setting this parameter to TFC_IGNORE_CTS causes the port to ignore the CTS signal. Setting this parameter to TFC_NONE causes the port to use the CTS signal, which is the default setting.
- `type` specifies the serial port type. It is one of RS232, RS232_MODEM, or RS485_2WIRE macros.
- `timeout` is not supported. This setting is ignored and is fixed at 600ms for backwards compatibility.

PID_DATA

The PID_DATA structure contains data for a PID control calculation. The structure contains input values, calculation results, and internal data that must be maintained from one execution to the next.

```
typedef struct pidData_type
{
   /* input values */
   float pv;
   float sp;
   float gain;
   float reset;
   float rate;
   float deadband;
   float fullScale;
   float zeroScale;
   float manualOutput;
   UINT32 period;
   BOOLEAN autoMode;

   /* calculation results */
   float output;
   BOOLEAN outOfDeadband;

   /* historic data values */
   float pvN1;
   float pvN2;
   float errorN1;
   UINT32 lastTime;
}
PID_DATA;
```

- `pv` is the process value
- `sp` is the set point
- `gain` is the gain
- `reset` is the reset time in seconds
- `rate` is the rate time in seconds
- `deadband` is the deadband
- `fullScale` is the full scale output limit
- `zeroScale` is the zero scale output limit
- `manualOutput` is the manual output value
- `period` is the execution period in milliseconds
- `autoMode` is the auto mode flag: `TRUE` = auto, `FALSE` = manual
- `output` is the last output value
- `outOfDeadband` is the error is outside the deadband
- `pvN1` is the process value from n-1 iteration
- `pvN2` is the process value from n-2 iteration
- `errorN1` is the error from n-1 iteration

- `lastTime` is the time of last execution

PROTOCOL_SETTINGS

The Extended Protocol Settings structure defines settings for a communication protocol. This structure differs from the standard settings in that it allows additional settings to be specified.

```
typedef struct protocolSettings_t
      {
      UCHAR type;
      UINT16 station;
      UCHAR priority;
      UINT16 SFMessaging;
      ADDRESS_MODE mode;
      }
      PROTOCOL_SETTINGS;
```

- `type` is the protocol type. It may be one of NO_PROTOCOL, MODBUS_RTU, or MODBUS_ASCII, AB_FULL_BCC, AB_FULL_CRC, AB_HALF_BCC, DNP or AB_HALF_CRC or PPP macros. When the type is set to PPP, the remaining settings in this structure are ignored. To set the remaining settings use the function **`mTcpSetInterfaceEx.`**
- `station` is the station address of the controller. Note that each serial port may have a different address. The valid values are determined by the communication protocol. This field is not used if the protocol type is NO_PROTOCOL.
- `priority` is the task priority of the protocol task. This field is not used if the protocol type is NO_PROTOCOL.
- `SFMessaging` is the enable Store and Forward messaging control flag.
- `ADDRESS_MODE` is the addressing mode, standard or extended.

PROTOCOL_SETTINGS_EX Type

This structure contains serial port protocol settings including Enron Modbus support.

```
typedef struct protocolSettingsEx_t
      {
      UCHAR type;
      UINT16 station;
      UCHAR priority;
      UINT16 SFMessaging;
      ADDRESS_MODE mode;
      BOOLEAN enronEnabled;
      UINT16 enronStation;
      }
      PROTOCOL_SETTINGS_EX;
```

- `type` is the protocol type. It may be one of NO_PROTOCOL, MODBUS_RTU, or MODBUS_ASCII, AB_FULL_BCC, AB_FULL_CRC, AB_HALF_BCC, DNP or AB_HALF_CRC or PPP macros. When the type is set to PPP, the remaining settings in this structure are ignored. To set the remaining settings use the function **`mTcpSetInterfaceEx.`**

- `station` is the station address of the controller. Note that each serial port may have a different address. The valid values are determined by the communication protocol. This field is not used if the protocol type is `NO_PROTOCOL`.
- `priority` is the task priority of the protocol task. This field is not used if the protocol type is `NO_PROTOCOL`.
- `SFMessaging` is the enable Store and Forward messaging control flag.
- `ADDRESS_MODE` is the addressing mode, `AM_standard` or `AM_extended`.
- `enronEnabled` determines if the Enron Modbus station is enabled. It may be `TRUE` or `FALSE`.
- `enronStation` is the station address for the Enron Modbus protocol. It is used if `enronEnabled` is set to `TRUE`. Valid values are 1 to 255 for standard addressing, and 1 to 65534 for extended addressing.

prot_settings

The Protocol Settings structure defines settings for a communication protocol. This structure differs from the extended settings in that it allows fewer settings to be specified.

```
struct prot_settings {
    UCHAR type;
    UCHAR station;
    UCHAR priority;
    UINT16 SFMessaging;
    };
```

- `type` is the protocol type. It may be one of NO_PROTOCOL, MODBUS_RTU, MODBUS_ASCII, AB_FULL_BCC, AB_HALF_BCC, AB_FULL_CRC, AB_HALF_CRC, DNP or PPP macros. When the type is set to PPP, the remaining settings in this structure are ignored. To set the remaining settings use the function **`mTcpSetInterfaceEx.`**
- `station` is the station address of the controller. Note that each serial port may have a different address. The valid values are determined by the communication protocol. This field is not used if the protocol type is NO_PROTOCOL.
- `priority` is the task priority of the protocol task. This field is not used if the protocol type is NO_PROTOCOL.
- `SFMessaging` is the enable Store and Forward messaging control flag.

prot_status

The `prot_status` structure contains protocol status information.

```
struct prot_status {
    UINT16 command_errors;
    UINT16 format_errors;
    UINT16 checksum_errors;
    UINT16 cmd_received;
    UINT16 cmd_sent;
    UINT16 rsp_received;
    UINT16 rsp_sent;
    UINT16 command;
    INT16  task_id;
    UINT16 stored_messages;
    UINT16 forwarded_messages;
    };
```

- `command_errors` is the number of messages received with invalid command codes.
- `format_errors` is the number of messages received with bad message data.
- `checksum_errors` is the number of messages received with bad checksums.
- `cmd_received` is the number of commands received.
- `cmd_sent` is the number of commands sent by the `master_message` function.
- `rsp_received` is the number of responses received by the `master_message` function.
- `rsp_sent` is the number of responses sent.
- `command` is the status of the last protocol command sent.
- `task_id` is the ID of the protocol task. This field is used by the `set_protocol` function to control protocol execution.
- `stored_messages` is the number of messages stored for forwarding.
- `forwarded_messages` is the number of messages forwarded.

PORT_CHARACTERISTICS

The PORT_CHARACTERISTICS type is a structure that contains serial port characteristics.

```
typedef struct portCharacteristics_tag {
      UINT16 dataflow;
      UINT16 buffering;
      UINT16 protocol;
      UINT32 options;
      } PORT_CHARACTERISTICS;
```

- `dataflow` is a bit mapped field describing the data flow options supported on the serial port. ANDing can isolate the options with the PC_FLOW_RX_RECEIVE_STOP, PC_FLOW_RX_XON_XOFF, PC_FLOW_TX_IGNORE_CTS or PC_FLOW_TX_XON_XOFF macros.
- `buffering` describes the buffering options supported. No buffering options are currently supported.
- `protocol` describes the protocol options supported. The macro, PC_PROTOCOL_RTU_FRAMING is the only option supported.
- `options` describes additional options supported. No additional options are currently supported.

pstatus

The pstatus structure contains serial port status information.

```
struct pstatus {
    UINT16 framing;
    UINT16 parity;
    UINT16 c_overrun;
    UINT16 b_overrun;
    UINT16 rx_buffer_size;
    UINT16 rx_buffer_used;
    UINT16 tx_buffer_size;
    UINT16 tx_buffer_used;
    UINT16 io_lines;
    };
```

- framing is the number of received characters with framing errors.
- parity is the number of received characters with parity errors.
- c_overrun is the number of received character overrun errors.
- b_overrun is the number of receive buffer overrun errors.
- rx_buffer_size is the size of the receive buffer in characters.
- rx_buffer_used is the number of characters in the receive buffer.
- tx_buffer_size is the size of the transmit buffer in characters.
- tx_buffer_used is the number of characters in the transmit buffer.
- io_lines is a bit mapped field indicating the status of the I/O lines on the serial port. The values for these lines differ between serial ports (see tables below). ANDing can isolate the signals with the SIGNAL_CTS, SIGNAL_DCD, SIGNAL_OH, SIGNAL_RING or SIGNAL_VOICE macros.

READSTATUS

The READSTATUS enumerated type indicates the status of an I^2C bus message read and may have one of the following values.

```
enum ReadStatus {
       RS_success,
       RS_selectFailed
       };
typedef enum ReadStatus READSTATUS;
```

- RS_success returns read was successful.
- RS_selectFailed returns slave device could not be selected

routingTable

The routingTable structure type describes an entry in the DNP Routing Table. This structure can be used with IP routing table entries but it cannot set the IP address. Use the dnpRoutingTableEx structure instead.

The DNP Routing Table is a list of routes, which are maintained in ascending order of DNP addresses.

```
typedef struct RoutingTable_type
```

```
{
      UINT16 address;   // station address
      UINT16 comPort;   // com port interface
      UINT16 retries;   // number of retries
      UINT16 timeout;   // timeout in milliseconds
} routingTable;
```

- `address` is the DNP station address of the destination station.
- `comPort` specifies the communications port interface. Allowed values are :
  1 = serial port com1
  2 = serial port com2
  3 = serial port com3
  4 = serial port com4
  103 = DNP over TCP, using LAN port
  104 = DNP over UDP, using LAN port
- `retries` is the number of times the data link layer will retry the message in the event of a failure.
- `timeout` is the timeout in milliseconds.

SF_TRANSLATION

The SF_TRANSLATION structure contains Store and Forward Messaging translation information. This is used to define an address and port translation.

```
typedef struct st_SFTranslationMTcp
{
   COM_INTERFACE slaveInterface;     // slave interface type
   UINT16        slaveStation;       // slave station address
   COM_INTERFACE forwardInterface;   // forwarding interface type
   UINT16        forwardStation;     // forwarding station address
   IP_ADDRESS    forwardIPAddress;   // forwarding IP address
}
SF_TRANSLATION;
```

- `slaveInterface` is the communication interface, which receives the slave command message. Valid interface types are: 1 = com1, 2 = com2, 3 = com3, 4= com4, 100 = Ethernet1.
- `slaveStation` is the station address used in the slave command message. Valid address range is: 0 to 255 in standard Modbus, 0 to 65534 in extended Modbus. 65535 = entry cleared. This station address must be different from the station address assigned to the `slaveInterface`.
- `forwardInterface` is the communication interface from which to forward the command message, as master. Valid interface types are: 1 = com1, 2 = com2, 3 = com3, 4= com4, 100 = Modbus/TCP network, 101 = Modbus RTU over UDP network, 102 = Modbus ASCII over UDP network.
- `forwardStation` is the station address of the remote slave device to forward the command message to. Valid address range is: 0 to 255 in standard Modbus, 0 to 65534 in extended Modbus. 65535 = entry cleared. This station address must be different from the station address assigned to the `forwardInterface`.
- `forwardIPAddress` is the IP address of the remote slave device to forward a Modbus IP command message to. Set to zero if not applicable.

SF_TRANSLATION_EX

The SF_TRANSLATION_EX structure contains Store and Forward Messaging translation information. This is used to define an address and port translation with a timeout.

```
typedef struct st_SFTranslationEx
{
      COM_INTERFACE slaveInterface;     // slave interface type
      UINT16        slaveStation;       // slave station address
      COM_INTERFACE forwardInterface;   // forwarding interface type
      UINT16        forwardStation;     // forwarding station address
      IP_ADDRESS    forwardIPAddress;   // forwarding IP address
      UINT16        timeout;            // time-out
}
SF_TRANSLATION_EX;
```

- `slaveInterface` is the communication interface which receives the slave command message. Valid interface types are: 1 = com1, 2 = com2, 3 = com3, 100 = Ethernet1.
- `slaveStation` is the station address used in the slave command message. Valid address range is: 0 to 255 in standard Modbus, 0 to 65534 in extended Modbus. 65535 = entry cleared. This station address must be different from the station address assigned to the `slaveInterface`.
- `forwardInterface` is the communication interface from which to forward the command message, as master. Valid interface types are: 1 = com1, 2 = com2, 3 = com3, 100 = Modbus/TCP network, 101 = Modbus RTU over UDP network, 102 = Modbus ASCII over UDP network.
- `forwardStation` is the station address of the remote slave device to forward the command message to. Valid address range is: 0 to 255 in standard Modbus, 0 to 65534 in extended Modbus. 65535 = entry cleared. This station address must be different from the station address assigned to the `forwardInterface`.
- `forwardIPAddress` is the IP address of the remote slave device to forward a Modbus IP command message to. Set to zero if not applicable.
- `timeout` is the maximum time the forwarding task waits for a valid response from the forward station, in tenths of seconds. Valid values are 0 to 65535.

SFTranslationStatus

The `SFTranslationStatus` structure contains information about a Store and Forward Translation table entry. It is used to report information about specific table entries.

```
struct SFTranslationStatus {
      UINT16 index;
      UINT16 code;
      };
```

- `index` is the location in the store and forward table to which the status code applies.
- `code` is the status code. It is one of SF_VALID, SF_INDEX_OUT_OF_RANGE, SF_NO_TRANSLATION, SF_PORT_OUT_OF_RANGE, SF_STATION_OUT_OF_RANGE, SF_ALREADY_DEFINED or SF_INVALID_FORWARDING_IP macros.

TASKINFO

The TASKINFO type is a structure containing information about a task.

```
/* Task Information Structure */
typedef struct taskInformation_tag {
   UINT16 taskID;
   UINT16 priority;
   UINT16 status;
   UINT16 requirement;
   UINT16 error;
   UINT16 type;
   } TASKINFO;
```

- `taskID` is the identifier of the task.
- `priority` is the execution priority of the task.
- `status` is the current execution status of the task. This may be one of TS_READY, TS_EXECUTING, TS_WAIT_ENVELOPE, TS_WAIT_EVENT, TS_WAIT_MESSAGE, or TS_WAIT_RESOURCE macros.
- `requirement` is used if the task is waiting for an event or resource. If the `status` field is TS_WAIT_EVENT, then `requirement` indicates on which event it is waiting. If the `status` field is TS_WAIT_RESOURCE then `requirement` indicates on which resource it is waiting.
- `error` is the task error code. This is the same value as returned by the `check_error` function.
- `type` is the task type. It will be either SYSTEM or APPLICATION.

taskInfo_tag

The `taskInfo_tag` structure contains start up task information.

```
struct taskInfo_tag {
        void *address;
        UINT16 stack;
        UINT16 identity;
        };
```

- `address` is the pointer to the start up routine.
- `stack` is the required stack size for the routine
- `identity` is the type of routine found (STARTUP_APPLICATION or STARTUP_SYSTEM)

TIME

The TIME structure contains time and date for reading or writing the real time clock.

```
struct clock {
    UINT16 year;
    UINT16 month;
    UINT16 day;
    UINT16 dayofweek;
    UINT16 hour;
    UINT16 minute;
    UINT16 second;
    } TIME;
```

- `year` is the current year. It is in the range 97 (for the year 1997) to 96 (for the year 2096).
- `month` is the current month. It is in the range 1 to 12.
- `day` is the current day. It is in the range 1 to 31.
- `dayofweek` is the current day of the week. It is in the range 1 to 7. The application program defines the meaning of this field.
- `hour` is the current hour. It is in the range 00 to 23.
- `minute` is the current minute. It is in the range 00 to 59.
- `second` is the current second. It is in the range 00 to 59.

timer_info

The `timer_info` structure contains information about a timer.

```
struct timer_info {
    UINT16 time;
    UINT16 interval;
    UINT16 interval_remaining;
};
```

- `time` is the time remaining in the timer in ticks.
- `interval` is the length of a timer tick in 10ths of a second.
- `interval_remaining` is the time remaining in the interval count down register in 10ths of a second.

timeval

struct timeval

{

long tv_sec; /* Number of Seconds */

long tv_usec; /* Number of micro seconds */

};

VERSION

The Firmware Version Information Structure holds information about the firmware.

```
typedef struct versionInfo_tag {
      UINT16 version;
      UINT16 controller;
      CHAR date[VI_DATE_SIZE + 1];
      CHAR copyright[VI_STRING_SIZE + 1];
      } VERSION;
```

- `version` is the firmware version number.
- `controller` is target controller for the firmware.
- `date` is a string containing the date the firmware was created.

- `copyright` is a string containing Control Microsystems copyright information.

WRITESTATUS

The WRITESTATUS enumerated type indicates the status of an I^2C bus message read and may have one of the following values.

```
enum WriteStatus {
       WS_success,
       WS_selectFailed,
       WS_noAcknowledge
       };
typedef enum WriteStatus WRITESTATUS;
```

- `WS_success` returns write was successful
- `WS_selectFailed` returns slave could not be selected
- `WS_noAcknowledge` returns slave failed to acknowledge data

Example Programs

Connecting with a Remote Controller Example

The following code shows how to connect to a remote controller using a modem. The example uses a US Robotics modem. It also demonstrates the use of the modemAbort function in an exit handler.

```
#include <ctools.h>
#include <string.h>

/* ----------------------------------------
   The shutdown function aborts any active
   modem connections when the task is ended.
   ---------------------------------------- */
void shutdown(void)
{
   modemAbort(com1);
}

int main(void)
{
   struct ModemSetup dialSettings;
   reserve_id portID;
   enum DialError status;
   enum DialState state;
   struct pconfig portSettings;
   TASKINFO taskStatus;

   /* Configure serial port 1 */
   portSettings.baud      = BAUD19200;
   portSettings.duplex    = FULL;
   portSettings.parity    = NONE;
   portSettings.data_bits = DATA8;
   portSettings.stop_bits = STOP1;
   portSettings.flow_rx   = RFC_MODBUS_RTU;
   portSettings.flow_tx   = TFC_NONE;
   portSettings.type      = RS232_MODEM;
   portSettings.timeout   = 600;
   request_resource(IO_SYSTEM);
   set_port(com1, &portSettings);
   release_resource(IO_SYSTEM);

   /* Configure US Robotics modem */
   dialSettings.port         = com1;
   dialSettings.dialAttempts = 3;
   dialSettings.detectTime   = 60;
   dialSettings.pauseTime    = 30;
   dialSettings.dialmethod   = 0;
   strcpy(dialSettings.modemCommand, "&F1 &A0 &K0 &M0 &B1");
   strcpy(dialSettings.phoneNumber, "555-1212");

   /* set up exit handler for this task */
   getTaskInfo(0, &taskStatus);
   installExitHandler(taskStatus.taskID, (FUNCPTR) shutdown);

   /* Connect to the remote controller */
   if (modemDial(&dialSettings, &portID) == DE_NoError)
```

```
    {
        do
        {
            /* Allow other tasks to execute */
            release_processor();

            /* Wait for initialization to complete */
            modemDialStatus(com1, portID, &status, &state);
        }
        while (state == DS_Calling);

        /* If the remote controller connected */
        if (state == DS_Connected)
        {
            /* Talk to remote controller here */
        }

        /* Terminate the connection */
        modemDialEnd(com1, portID, &status);
    }
}
```

Note that a pause of a few seconds is required between terminating a connection and initiating a new call. This pause allows the external modem time to hang up.

Create Task Example

```
#include <ctools.h>

#define   TIME_TO_PRINT    20

void task1(void)
{
       int a, b;

       while (TRUE)
       {
          /* body of task 1 loop - processing I/O */

          request_resource(IO_SYSTEM);
          a = dbase(MODBUS, 30001);
          b = dbase(MODBUS, 30002);
          setdbase(MODBUS, 40020, a * b);
          release_resource(IO_SYSTEM);

          /* Allow other tasks to execute */
          release_processor();
       }
}

void task2(void)
{
       while(TRUE)
       {
          /* body of task 2 loop - event handler */
          wait_event(TIME_TO_PRINT);
          fprintf(com1,"It's time for a coffee break\r\n");
       }
}
/* ------------------------------------------
   The shutdown function stops the signalling
   of TIME_TO_PRINT events when application is
       stopped.
   ------------------------------------------ */
void shutdown(void)
{
       endTimedEvent(TIME_TO_PRINT);
}
int main(void)
{
       TASKINFO taskStatus;

       /* continuos processing task at priority 100 */
       create_task(task1, 100, APPLICATION, 2);

       /* event handler needs larger stack for printf function */
       create_task(task2, 75, APPLICATION, 4);

       /* set up task exit handler to stop
       signalling of events when this task ends */
       getTaskInfo(0, &taskStatus);
       installExitHandler(taskStatus.taskID, (FUNCPTR) shutdown);

       /* start timed event to occur every 10 sec */
       startTimedEvent(TIME_TO_PRINT, 100);
```

```
        interval(0, 10);
        while(TRUE)
        {
           /* body of main task loop */
           /* other processing code */
           /* Allow other tasks to execute */
           release_processor();
        }
}
```

DNP Configuration Example

```
/* ------------------------------------------------------------------
   SCADAPack2 C++ Application Main Program
   Copyright 2001 - 2004, Control Microsystems Inc.

   The following program demonstrates how to configure DNP for operation
   on com3 of the SCADAPack2.

   ------------------------------------------------------------------ */
#include <ctools.h>

/* ------------------------------------------------------------------
   C++ Function Prototypes
   ------------------------------------------------------------------ */
// add prototypes here

/* ------------------------------------------------------------------
   C Function Prototypes
   ------------------------------------------------------------------ */
extern "C"
{
      // add prototypes here
}
UINT32 mainPriority = 100;
UINT32 mainStack = 4;
UINT32 applicationGroup = 0;

/* ------------------------------------------------------------------
   main

   This routine is the main application loop.
   ------------------------------------------------------------------ */
int main(void)
{
   //-------------------------------------------------------------------
   // Variable declaration
   //-------------------------------------------------------------------
       UINT16                 index;              // loop index
        PROTOCOL_CONFIGURATION    protocolSettings;   // protocol
settings
      dnpConfiguration       configuration;      // configuration
settings
      dnpBinaryOutput     binaryOutput;    // binary output settings
      dnpBinaryInput         binaryInput;     // binary input settings
      dnpAnalogInput         analogInput;     // analog input settings
      dnpAnalogOutput     analogOutput;    // analog output settings
      dnpCounterInput        counterInput;    // conter input settings
   //-------------------------------------------------------------------
   // Stop any protocol currently active on com port 3
   //-------------------------------------------------------------------
       get_protocol(com3, &protocolSettings);
       protocolSettings.type = NO_PROTOCOL;
       set_protocol(com3, &protocolSettings);
   //-------------------------------------------------------------------
   // Load DNP Configuration Parameters
   //-------------------------------------------------------------------
       configuration.masterAddress      = 100;
       configuration.rtuAddress         = 1;
       configuration.datalinkConfirm    = FALSE;
       configuration.datalinkRetries    = DEFAULT_DLINK_RETRIES;
       configuration.datalinkTimeout    = DEFAULT_DLINK_TIMEOUT;
```

```
    configuration.operateTimeout     = DEFAULT_OPERATE_TIMEOUT;
    configuration.applicationConfirm = FALSE;
    configuration.maximumResponse    = DEFAULT_MAX_RESP_LENGTH;
    configuration.applicationRetries = DEFAULT_APPL_RETRIES;
    configuration.applicationTimeout = DEFAULT_APPL_TIMEOUT;
    configuration.timeSynchronization = NO_TIME_SYNC;
    configuration.BI_number          = 1701;
    configuration.BI_startAddress    = 0;
    configuration.BI_reportingMethod = REPORT_ALL_EVENTS;
    configuration.BI_soeBufferSize   = 1000;
    configuration.BO_number          = 1051;
    configuration.BO_startAddress    = 0;
    configuration.CI16_number        = 50;
    configuration.CI16_startAddress  = 0;
    configuration.CI16_reportingMethod  = REPORT_ALL_EVENTS;
    configuration.CI16_bufferSize    = 0;
    configuration.CI32_number        = 0;
    configuration.CI32_startAddress  = 100;
    configuration.CI32_reportingMethod  = REPORT_ALL_EVENTS;
    configuration.CI32_bufferSize    = 0;
    configuration.CI32_wordOrder     = MSW_FIRST;
    configuration.AI16_number        = 751;
    configuration.AI16_startAddress  = 0;
    configuration.AI16_reportingMethod  = REPORT_ALL_EVENTS;
    configuration.AI16_bufferSize    = 1000;
    configuration.AI32_number        = 0;
    configuration.AI32_startAddress  = 100;
    configuration.AI32_reportingMethod  = REPORT_ALL_EVENTS;
    configuration.AI32_bufferSize    = 0;
    configuration.AI32_wordOrder     = MSW_FIRST;
    configuration.AISF_number        = 0;
    configuration.AISF_startAddress  = 200;
    configuration.AISF_reportingMethod  = REPORT_CHANGE_EVENTS;
    configuration.AISF_bufferSize    = 0;
    configuration.AISF_wordOrder     = MSW_FIRST;
    configuration.AO16_number        = 20;
    configuration.AO16_startAddress  = 0;
    configuration.AO32_number        = 12;
    configuration.AO32_startAddress  = 100;
    configuration.AO32_wordOrder     = MSW_FIRST;
    configuration.AOSF_number        = 0;
    configuration.AOSF_startAddress  = 200;
    configuration.AOSF_wordOrder     = MSW_FIRST;
    configuration.autoUnsolicitedClass1 = TRUE;
    configuration.holdTimeClass1     = 10;
    configuration.holdCountClass1    = 3;
    configuration.autoUnsolicitedClass2 = TRUE;
    configuration.holdTimeClass2     = 10;
    configuration.holdCountClass2    = 3;
    configuration.autoUnsolicitedClass3 = TRUE;
    configuration.holdTimeClass3     = 10;
    configuration.holdCountClass3    = 3;
    configuration.enableUnsolicitedOnStartup = TRUE;
    configuration.sendUnsolicitedOnStartup   = FALSE;
    configuration.level2Compliance           = FALSE;
//---------------------------------------------------------------------
// Set DNP Configuration
//---------------------------------------------------------------------
    dnpSaveConfiguration(&configuration);
    //-------------------------------------------------------------
    // Start DNP protocol on com port 3
    //-------------------------------------------------------------
    get_protocol(com3, &protocolSettings);
```

```
        protocolSettings.type = DNP;
        set_protocol(com3, &protocolSettings);
        //-------------------------------------------------------------
        // Configure Binary Output Points
        //-------------------------------------------------------------
        for (index = 0; index < configuration.BO_number; index++)
        {
           binaryOutput.modbusAddress1 = 1 + index;
           binaryOutput.modbusAddress2 = 1 + index;
           binaryOutput.controlType = NOT_PAIRED;
           dnpSaveBOConfig(configuration.BO_startAddress + index,
&binaryOutput);
        }
        //-------------------------------------------------------------
        // Configure Binary Input Points
        //-------------------------------------------------------------
        for (index = 0; index < configuration.BI_number; index++)
        {
           binaryInput.modbusAddress = 10001 + index;
           binaryInput.eventClass = CLASS_1;
           dnpSaveBIConfig(configuration.BI_startAddress + index,
&binaryInput);
        }
        //-------------------------------------------------------------
        // Configure 16 Bit Analog Input Points
        //-------------------------------------------------------------
        for (index = 0; index < configuration.AI16_number; index++)
        {
           analogInput.modbusAddress = 30001 + index;
           analogInput.eventClass = CLASS_2;
           analogInput.deadband = 1;
           dnpSaveAI16Config(configuration.AI16_startAddress + index,
&analogInput);
        }
        //-------------------------------------------------------------
        // Configure 32 Bit Analog Input Points
        //-------------------------------------------------------------
        for (index = 0; index < configuration.AI32_number; index++)
        {
           analogInput.modbusAddress = 30001 + index;
           analogInput.eventClass = CLASS_2;
           analogInput.deadband = 1;
           dnpSaveAI32Config(configuration.AI16_startAddress + index,
&analogInput);
        }
        //-------------------------------------------------------------
        // Configure 16 Bit Analog Output Points
        //-------------------------------------------------------------
        for (index = 0; index < configuration.AO16_number; index++)
        {
           analogOutput.modbusAddress = 40001 + index;
           dnpSaveAO16Config(configuration.AO16_startAddress + index,
&analogOutput);
        }
        //-------------------------------------------------------------
        // Configure 32 Bit Analog Output Points
        //-------------------------------------------------------------
        for (index = 0; index < configuration.AO32_number; index++)
        {
           analogOutput.modbusAddress = 41001 + index * 2;
           dnpSaveAO32Config(configuration.AO32_startAddress + index,
&analogOutput);
        }
```

```
        //---------------------------------------------------------------
        // Configure 16 Bit Counter Input Points
        //---------------------------------------------------------------
        for (index = 0; index < configuration.CI16_number; index++)
        {
           counterInput.modbusAddress = 30001 + index;
           counterInput.eventClass = CLASS_3;
           counterInput.threshold = 1;
           dnpSaveCI16Config(configuration.CI16_startAddress + index,
&counterInput);
        }
        //---------------------------------------------------------------
        // Configure 32 Bit Counter Input Points
        //---------------------------------------------------------------
        for (index = 0; index < configuration.CI32_number; index++)
        {
           counterInput.modbusAddress = 30001 + index * 2;
           counterInput.eventClass = CLASS_3;
           counterInput.threshold = 1;
           dnpSaveCI32Config(configuration.CI32_startAddress + index,
&counterInput);
        }

        // main loop
        while (TRUE)
        {
           // add remainder of program here

           // release processor to other priority 254 tasks
           release_processor();
        }
}
```

Get Program Status Example

This program stores a default alarm limit into the I/O database the first time it is run. On subsequent executions, it uses the limit in the database. The limit in the database can be modified by a communication protocol during execution.

```
#include <ctools.h>

#define HI_ALARM    41000
#define ALARM_OUTPUT   1026
#define SCAN_EVENT     0

int main( void )
{
   if (getProgramStatus((FUNCPTR)main) == NEW_PROGRAM)
   {
      /* Set default alarm limit */
      request_resource(IO_SYSTEM);
      setdbase(MODBUS, HI_ALARM, 4000);
      release_resource(IO_SYSTEM);

      /* Use values in database from now on */
      setProgramStatus((FUNCPTR)main, PROGRAM_EXECUTED);
   }
   while (TRUE)
   {
      INT16  ain[8];   // analog input module data

      request_resource(IO_SYSTEM);

      /* Scan ain module */
      ioRequest(MT_Ain8, 0);
      ioNotification(SCAN_EVENT);
      wait_event(SCAN_EVENT);
      ioReadAin8(0, ain);

      /* Test input against alarm limits */
      if (ain[0] > dbase(MODBUS, HI_ALARM))
         setdbase(MODBUS, ALARM_OUTPUT, 1);
      else
         setdbase(MODBUS, ALARM_OUTPUT, 0);

      release_resource(IO_SYSTEM);

      /* Allow other tasks to execute */
      release_processor();
   }
}
```

Get Task Status Example

The following program displays information about all valid tasks.

```
#include <string.h>
#include <ctools.h>

int main(void)
{
        struct prot_settings settings;
        TASKINFO taskStatus;
        unsigned task;
        char state[6][20];
        char type[2][20];

        /* Set up state strings */
        strcpy(state[TS_READY], "Ready");
        strcpy(state[TS_EXECUTING], "Executing");
        strcpy(state[TS_WAIT_ENVELOPE], "Waiting for Envelope");
        strcpy(state[TS_WAIT_EVENT], "Waiting for Event");
        strcpy(state[TS_WAIT_MESSAGE], "Waiting for Message");
        strcpy(state[TS_WAIT_RESOURCE], "Waiting for Resource");

        /* Set up type strings */
        strcpy(type[APPLICATION], "Application");
        strcpy(type[SYSTEM], "System");

        /* Disable the protocol on serial port 1 */
        settings.type = NO_PROTOCOL;
        settings.station = 1;
        settings.priority = 250;
        settings.SFMessaging = FALSE;
        request_resource(IO_SYSTEM);
        set_protocol(com1, &settings);
        release_resource(IO_SYSTEM);

        /* display information about all tasks */
        for (task = 0; task <= RTOS_TASKS; task++)
        {
            if (getTaskInfo(task, &taskStatus))
            {
                /* show information for valid task */
                fprintf(com1, "\r\n\r\nInformation about task %d:\r\n", task);
                fprintf(com1, "    Task ID:  %d\r\n", taskStatus.taskID);
                fprintf(com1, "    Priority: %d\r\n", taskStatus.priority);
                fprintf(com1, "    Status:   %s\r\n", state[taskStatus.status]);
                if (taskStatus.status == TS_WAIT_EVENT)
                {
                    fprintf(com1, "    Event:    %d\r\n", taskStatus.requirement);
                }
                if (taskStatus.status == TS_WAIT_RESOURCE)
                {
                    fprintf(com1, "    Resource: %d\r\n", taskStatus.requirement);
                }
                fprintf(com1, "    Error:    %d\r\n", taskStatus.error);
                fprintf(com1, "    Type:     %s\r\n", type[taskStatus.type]);
            }
        }

        while (TRUE)
        {
            /* Allow other tasks to execute */
            release_processor();
        }
}
```

Handler Function Example

```
/* ---------------------------------------------
   handler.c

   This is a sample program for the  InstallModbusHandler function. This
sample    program uses function code 71 to  demonstrate a    simple method
for using the    installModbusHandler function.
   When the handler is installed Modbus ASCII  messages using function
code 71 that are    received on com2 of the controller will
   be processed as shown in the program text.

   To turn on digital output 00001:
   From a terminal send the ASCII command  :014701B7
   Where;
       01 is the station address
       47 is the function code in hex
       01 is the command for the function code
       B7 is the message checksum

   To turn off digital output 00001:
   From a terminal send the ASCII command  :014700B8
   Where;
       01 is the station address
       47 is the function code in hex
       00 is the command for the function code
       B8 is the message checksum
   ------------------------------------------- */
#include <ctools.h>

static unsigned myModbusHandler(
       UCHAR * message,
       UINT16    messageLength,
       UCHAR * response,
       UINT16 * responseLength
       )
{
       UCHAR * pMessage;
       UCHAR * pResponse;

       pMessage = message;

       if (*pMessage == 71)
       {
          /* Action for command data */
          pMessage++;

           if (*pMessage == 0)
          {
             request_resource(IO_SYSTEM);
             setdbase(MODBUS, 1, 0);
             release_resource(IO_SYSTEM);

             pResponse = response;

             *pResponse   = 71;
             pResponse++;
             *pResponse   = 'O';
             pResponse++;
             *pResponse   = 'F';
```

```
            pResponse++;
            *pResponse   = 'F';
            pResponse++;

            *responseLength = 4;

            return NORMAL;
         }
         else if (*pMessage == 1)
         {
            request_resource(IO_SYSTEM);
            setdbase(MODBUS, 1, 1);
            release_resource(IO_SYSTEM);

            pResponse = response;
            *pResponse   = 71;
            pResponse++;
            *pResponse   = 'O';
            pResponse++;
            *pResponse   = 'N';
            pResponse++;
            *responseLength = 3;

            return NORMAL;
         }
         else
         {
            return FUNCTION_NOT_HANDLED;
         }
      }
      else
      {
         return FUNCTION_NOT_HANDLED;
      }
}

static void shutdown(void)
{
      removeModbusHandler(myModbusHandler);
}

/* ------------------------------------------------
   main

   This routine is the modbus slave application.
   Serial port com2 is configured for Modbus ASCII   protocol.
   Register Assignment is configured.
   The modbus handler is installed.
   The exit handler is installed.
   ------------------------------------------- */
int main(void)
{
      TASKINFO taskStatus;

      struct pconfig portSettings;
      struct prot_settings protSettings;

      portSettings.baud      = BAUD9600;
      portSettings.duplex    = FULL;
      portSettings.parity    = NONE;
      portSettings.data_bits = DATA7;
      portSettings.stop_bits = STOP1;
      portSettings.flow_rx   = RFC_NONE;
```

```
      portSettings.flow_tx   = TFC_NONE;
      portSettings.type      = RS232;
      portSettings.timeout   = 600;
      set_port(com2, &portSettings);

      get_protocol(com2, &protSettings);
      protSettings.station     = 1;
      protSettings.type       = MODBUS_ASCII;
      set_protocol(com2, &protSettings);

      /* Configure Register Assignment */
      clearRegAssignment();
      addRegAssignment(DIN_generic8, 0, 10017, 0, 0, 0);
      addRegAssignment(DIAG_protocolStatus,1,31000, 0, 0, 0);

   /* Install Exit Handler */
      getTaskInfo(0, &taskStatus);
      installExitHandler(taskStatus.taskID, (FUNCPTR) shutdown);

      /* Install Modbus Handler */
      request_resource(IO_SYSTEM);
      installModbusHandler(myModbusHandler);
      release_resource(IO_SYSTEM);

      while(TRUE)
      {
         release_processor();
      }
}
```

Install Serial Port Handler Example

```
#include <ctools.h>

#define CHAR_RECEIVED 11


/* ------------------------------------------
   signal

   This routine signals an event when a character
   is received on com1. If there is an error, the
   character is ignored.
   ------------------------------------------ */

void signal(UINT16 character, UINT16 error)
{
       if (error == 0)
          interrupt_signal_event( CHAR_RECEIVED );

       character;
}

/* ------------------------------------------
   main

   This program displays all characters received
   on com1 using an installed handler to signal
   the reception of a character.
   ------------------------------------------ */

int main(void)
{
       struct prot_settings protocolSettings;
       int character;

       /* Disable protocol */
       get_protocol(com1, &protocolSettings);
       protocolSettings.type = NO_PROTOCOL;
       request_resource(IO_SYSTEM);
       set_protocol(com1, &protocolSettings);
       release_resource(IO_SYSTEM);

       /* Enable character handler */
       install_handler(com1, signal);

       /* Print each character as it is received */
       while (TRUE)
       {
          wait_event(CHAR_RECEIVED);
          character = fgetc(com1);
          if (character == EOF)
          {
             // clear overflow error flag to re-enable com1
             clearerr(com1);
          }
          fputs("character: ", com1);
          fputc(character, com1);
          fputs("\r\n", com1);
       }
```

```
}
```

Install Clock Handler Example

```
/* ------------------------------------------
   This program demonstrates how to call a
   function at a specific time of day.
   ------------------------------------------ */

#include <ctools.h>

#define   ALARM_EVENT     20

/* ------------------------------------------
       This function signals an event when the alarm
       occurs.
------------------------------------------ */
void alarmHandler(void)
{
       interrupt_signal_event( ALARM_EVENT );
}

/* ------------------------------------------
   This task processes alarms signaled by the
       clock handler
------------------------------------------ */
void processAlarms(void)
{
       while(TRUE)
       {
          wait_event(ALARM_EVENT);

          /* Reset the alarm for the next day */
          request_resource(IO_SYSTEM);
          resetClockAlarm();
          release_resource(IO_SYSTEM);

          fprintf(com1, "It's quitting time!\r\n");
       }
}

int main(void)
{
       struct prot_settings settings;
       ALARM_SETTING alarm;

       /* Disable the protocol on serial port 1 */
       settings.type = NO_PROTOCOL;
       settings.station = 1;
       settings.priority = 250;
       settings.SFMessaging = FALSE;
       request_resource(IO_SYSTEM);
       set_protocol(com1, &settings);
       release_resource(IO_SYSTEM);

       /* Install clock handler function */
       installClockHandler(alarmHandler);

       /* Create task for processing alarm events */
       create_task(processAlarms, 75, APPLICATION, 4);

       /* Set real time clock alarm */
```

```
        alarm.type   = AT_ABSOLUTE;
        alarm.hour   = 16;
        alarm.minute = 0;
        alarm.second = 0;

        request_resource(IO_SYSTEM);
        setClockAlarm(alarm);
        release_resource(IO_SYSTEM);

        while(TRUE)
        {
           /* body of main task loop */

           /* other processing code */

           /* Allow other tasks to execute */
           release_processor();
        }
}
```

Install Database Handler Example

This program assumes that the pointer pAllocatedMem has been declared in nvMemory.h.

```
/* ------------------------------------------------------------
   This is a sample ISaGRAF application for the
   installDbaseHandler and installSetdbaseHandler functions.
   This sample program demonstrates database handlers for the
   Modbus registers:

        1001 to  1100
       11001 to 11100
       31001 to 31100
       41001 to 41100

   This database is allocated in non-volatile memory.

   When the handlers are installed, calls to the functions dbase,
   setdbase, databaseRead, and databaseWrite for these Modbus
   registers will call these handlers. This is true as long as
   the register is not already assigned to an ISaGRAF variable.

   Note that these database access functions are used by C++
   applications and by all protocols.
   ------------------------------------------------------------ */
#include <ctools.h>
#include <string.h>
#include “nvMemory.h”

#define SAMPLE_SIZE    100
#define SCAN_EVENT_NO  0

// custom Modbus database structure
struct myDatabase
{
   BOOLEAN coilDbase[SAMPLE_SIZE];
   BOOLEAN statusDbase[SAMPLE_SIZE];
   INT16 inputDbase[SAMPLE_SIZE];
   INT16 holdingDbase[SAMPLE_SIZE];
};

#define MEM_SIZE (sizeof(struct myDatabase))

/* ------------------------------------------------------------
   This is the dbase handler.
   ------------------------------------------------------------ */
static BOOLEAN dbaseHandler(
   UINT16 address,  /* Modbus register address */
   INT16  *value       /* pointer to value at address */
   )
{
   struct myDatabase * pMyDatabase; // pointer to custom database

   pMyDatabase = (struct myDatabase *) pNvMemory->pAllocatedMem;
   if (pMyDatabase == NULL)
   {
      // database could not be allocated
      return FALSE;
   }

   if ((address > 1000) && (address <= 1000 + SAMPLE_SIZE))
```

```
    {
        *value = pMyDatabase->coilDbase[address - 1001];
        return TRUE;
    }
    else if ((address > 11000)&&(address <= 11000 + SAMPLE_SIZE))
    {
        *value = pMyDatabase->statusDbase[address - 11001];
        return TRUE;
    }
    else if ((address > 31000)&&(address <= 31000 + SAMPLE_SIZE))
    {
        *value = pMyDatabase->inputDbase[address - 31001];
        return TRUE;
    }
    else if ((address > 41000)&&(address <= 41000 + SAMPLE_SIZE))
    {
        *value = pMyDatabase->holdingDbase[address - 41001];
        return TRUE;
    }
    else
    {
        /* all other addresses are not handled */
        return FALSE;
    }
}

/* ------------------------------------------------------------
   This is the setdbase handler.
   ------------------------------------------------------------ */
static BOOLEAN setdbaseHandler(
    UINT16 address,  /* Modbus register address */
    INT16  value     /* value to write at address */
    )
{
    struct myDatabase * pMyDatabase; // pointer to custom database

    pMyDatabase = (struct myDatabase *) pNvMemory->pAllocatedMem;
    if (pMyDatabase == NULL)
    {
        // database could not be allocated
        return FALSE;
    }

    if ((address > 1000) && (address <= 1000 + SAMPLE_SIZE))
    {
        if (value == 0)
        {
            pMyDatabase->coilDbase[address - 1001] = FALSE;
        }
        else
        {
            pMyDatabase->coilDbase[address - 1001] = TRUE;
        }
        return TRUE;
    }
    else if ((address > 11000) && (address <= 11000 + SAMPLE_SIZE))
    {
        if (value == 0)
        {
            pMyDatabase->statusDbase[address - 11001] = FALSE;
        }
        else
        {
```

```
            pMyDatabase->statusDbase[address - 11001] = TRUE;
         }
         return TRUE;
      }
      else if ((address > 31000)&&(address <= 31000 + SAMPLE_SIZE))
      {
         pMyDatabase->inputDbase[address - 31001] = value;
         return TRUE;
      }
      else if ((address > 41000)&&(address <= 41000 + SAMPLE_SIZE))
      {
         pMyDatabase->holdingDbase[address - 41001] = value;
         return TRUE;
      }
      else
      {
         /* all other addresses are not handled */
         return FALSE;
      }
   }

   /* ------------------------------------------------------------
      This is the exit handler.
      ------------------------------------------------------------ */
   static void shutdown(void)
   {
      /* remove database handlers */
      installDbaseHandler(NULL);
      installSetdbaseHandler(NULL);
   }

   /* ------------------------------------------------------------
      This routine initializes the custom database.
      The database memory is allocated if application has just been
      downloaded. The exit handler is installed and the database
      handlers are installed.
      ------------------------------------------------------------ */
   static void initializeDatabase(void)
   {
      TASKINFO taskStatus;
      BOOLEAN  status;

      if (getProgramStatus((FUNCPTR)main) == NEW_PROGRAM)
      {
         // Application has just been downloaded. Any memory
         // previously allocated has been freed automatically.
         // Allocate non-volatile dynamic memory.
         request_resource(DYNAMIC_MEMORY);
         status = allocateMemory((void **)&(pNvMemory-
   >pAllocatedMem), MEM_SIZE);
         release_resource(DYNAMIC_MEMORY);
         if (status == TRUE)
         {
            // set program status so memory is not re-allocated
            // until next program download
            setProgramStatus((FUNCPTR)main, PROGRAM_EXECUTED);

            // zero-fill new custom database
            memset(pNvMemory->pAllocatedMem, 0, MEM_SIZE);
         }
         else
         {
            // memory could not be allocated
```

```
            pNvMemory->pAllocatedMem = NULL;
         }
      }

      // install exit handler to remove the custom database
      // if the application is stopped or erased
      getTaskInfo(0, &taskStatus);
      installExitHandler(taskStatus.taskID, (FUNCPTR) shutdown);

      // install database handlers
      installDbaseHandler(dbaseHandler);
      installSetdbaseHandler(setdbaseHandler);
   }

   /* -------------------------------------------------------------
      This routine is the main program. The custom i/o database is
      initialized. The database is then updated continuously with
      I/O data in the main loop.
      ------------------------------------------------------------- */
   int main(void)
   {
      UINT16 dinData;         // data from 16 Din points
      INT16  ainData[8];      // data from 8 Ain points
      UINT16 doutData = 0;    // data written to Dout points
      UINT16 index;

      // initialize custom i/o database
      initializeDatabase();

      // main loop
      while (TRUE)
      {
         // write data to Output tables
         ioWrite5601Outputs(doutData);

         // add I/O requests to the I/O System queue
         ioRequest(MT_5601Inputs, 0);
         ioRequest(MT_5601Outputs, 0);
         // this event signals completion of preceding i/o requests
         ioNotification(SCAN_EVENT_NO);
         // wait for your event to be signalled when all your
         // I/O requests have been processed.
         wait_event(SCAN_EVENT_NO);

         // get the data read from Input modules
         ioRead5601Inputs(dinData, ainData);

         request_resource(IO_SYSTEM);

         // copy Ain data to the database
         for (index=0; index<8; index++)
         {
            setdbase(MODBUS, 31001 + index, ainData[index]);
         }

         // copy Din data to the database
         for (index=0; index<16; index++)
         {
            if (dinData & 0x01)
            {
               setdbase(MODBUS, 11001 + index, 1);
            }
            else
```

```
            {
                setdbase(MODBUS, 11001 + index, 0);
            }
            dinData >>= 1;
        }

        // get 12 DOUT points from the database
        for (index=0; index<12; index++)
        {
            doutData <<= 1;
            if (dbase(MODBUS, 1012 - index))
            {
                doutData |= 1;
            }
        }

        release_resource(IO_SYSTEM);

        // release processor to other priority 254 tasks
        release_processor();
    }
}
```

Memory Allocation Example

This program allocates dynamic non-volatile memory only when the C++ Application is run the first time after downloading.

Refer to the section *Non-Volatile Data Sections* for instructions on declaring non-volatile variables. This program assumes that the pointer pAllocatedMem has been declared in nvMemory.h.

```
#include <ctools.h>
#include “nvMemory.h"

struct    myTable
{
   UINT32 data[100];
};

#define MEM_SIZE  (sizeof(struct myTable))

int main( void )
{
   BOOLEAN status;
   struct myTable * pTable;

   status = TRUE;
   if (getProgramStatus((FUNCPTR)main) == NEW_PROGRAM)
   {
     // Application has just been downloaded.
     request_resource(DYNAMIC_MEMORY);
     status = allocateMemory((void **)&(pNvMemory->pAllocatedMem),
   MEM_SIZE);
     release_resource(DYNAMIC_MEMORY);

     if (status == TRUE)
       {
           // set program status so memory is not re-allocated
           // until application is downloaded again
           setProgramStatus((FUNCPTR)main, PROGRAM_EXECUTED);
     }
   }

   // use non-volatile memory for table structure
   pTable = (struct myTable *) (pNvMemory->pAllocatedMem);

   while (TRUE)
   {
     if (status == TRUE)
     {
          // pTable is used in remainder of program
          // ...
     }
     else
     {
          // print error message
     }

     // Allow other tasks to execute
     release_processor();
   }
}
```

Master Message Example Using Modbus Protocol

This program sends a master message, on **com2**, using the Modbus protocol, then waits for a response from the slave. The number of good and failed messages is printed to **com1**.

```
/* ----------------------------------------
   poll.c
   Polling program for Modbus slave.
   ---------------------------------------- */

#include <ctools.h>

/* ----------------------------------------
   wait_for_response

   The wait_for_response function waits for a
   response to be received to a master_message on
   the serial port specified by stream. It returns
   when a response is received, or when the period
   specified by time (in tenths of a second)
   expires.
   ---------------------------------------- */

void wait_for_response(UCHAR port, unsigned time)
{
       UINT32 startTime;
       struct prot_status status;
       static unsigned long good, bad;

       startTime = readStopwatch();
       do {
          /* Allow other tasks to execute */
          release_processor();

          status = get_protocol_status(stream);
       }
       while ((readStopwatch() – startTime) < (100 * time) &&
status.command == MM_SENT);

       if (status.command == MM_RECEIVED)
          good++;
       else
          bad++;
       fprintf(com1, "Good: %8lu  Bad: %8lu\r", good,
       bad);
}
/* ----------------------------------------
       main

   The main function sets up serial ports then
   sends commands to a Modbus slave.
   ---------------------------------------- */
int main(void)
{
       struct prot_settings settings;
       struct pconfig portset;

       request_resource(IO_SYSTEM);

       /* disable protocol on serial port 1 */
       settings.type = NO_PROTOCOL;
```

```
    settings.station = 1;
    settings.priority = 250;
    settings.SFMessaging = FALSE;
    set_protocol(com1, &settings);

    /* Set communication parameters for port 1 */
    portset.baud      = BAUD9600;
    portset.duplex    = FULL;
    portset.parity    = NONE;
    portset.data_bits = DATA8;
    portset.stop_bits = STOP1;
    portset.flow_rx   = RFC_NONE;
    portset.flow_tx   = TFC_NONE;
    portset.type      = RS232;
    portset.timeout   = 600;
    set_port(com1, &portset);

    /* enable Modbus protocol on serial port 2 */
    settings.type = MODBUS_ASCII;
    settings.station = 2;
    settings.priority = 250;
    settings.SFMessaging = FALSE;
    set_protocol(com2, &settings);

    /* Set communication parameters for port 2 */
    portset.baud      = BAUD9600;
    portset.duplex    = HALF;
    portset.parity    = NONE;
    portset.data_bits = DATA8;
    portset.stop_bits = STOP1;
    portset.flow_rx   = RFC_NONE;
    portset.flow_tx   = TFC_NONE;
    portset.type      = RS485_4WIRE;
    portset.timeout   = 600;
    set_port(com2, &portset);

    release_resource(IO_SYSTEM);

    /* enable timers used in wait_for_response */
    runTimers(TRUE);

    /* Main communication loop */
    while (TRUE)
    {
       /* Transfer slave inputs to outputs */
       request_resource(IO_SYSTEM);
       master_message(com2, 2, 1, 10001, 17, 8);
       release_resource(IO_SYSTEM);
       wait_for_response(com2, 10);

       /* Transfer inputs to slave outputs */
       request_resource(IO_SYSTEM);
       master_message(com2, 15, 1, 1, 10009, 8);
       release_resource(IO_SYSTEM);
       wait_for_response(com2, 10);

       /* Allow other tasks to execute */
       release_processor();
    }
}
```

Master Message Example Using serialModbusMaster

This program sends master messages on com2 demonstrating two methods using the function serialModbusMaster.

```
/* ------------------------------------------
   SCADAPack2 C++ Application Main Program
   Copyright 2001 - 2004, Control Microsystems Inc.
   ------------------------------------------ */
#include <ctools.h>

// function prototypes
static void master2(void);

/* ------------------------------------------
   Modular variables
   ------------------------------------------ */

// declare session as modular to reduce stack space usage
static MODBUS_SESSION  masterSession1;
static MODBUS_SESSION  masterSession2;

/* ------------------------------------------
   main

   The main function sets up serial port then
   sends commands to a Modbus slave. This task
   monitors the command status to check when
   the response is received. This method is
   useful when other processing can be done
   while waiting for the response.
   ------------------------------------------ */

UINT32 mainPriority = 100;
UINT32 mainStack = 4;
UINT32 applicationGroup = 0; int main(void)
{
   MASTER_MESSAGE   message;
   BOOLEAN          status;
   UINT16           good, bad;
   struct prot_settings settings;
   struct pconfig portset;

   request_resource(IO_SYSTEM);

   // enable Modbus protocol on com2
   settings.type = MODBUS_RTU;
   settings.station = 1;
   settings.priority = 250;
   settings.SFMessaging = FALSE;
   set_protocol(com2, &settings);

   // set communication parameters for com2
   portset.baud = BAUD9600;
   portset.duplex = FULL;
   portset.parity = NONE;
   portset.data_bits = DATA8;
   portset.stop_bits = STOP1;
   portset.flow_rx = RFC_MODBUS_RTU;
   portset.flow_tx = TFC_NONE;
   portset.type = RS232;
   portset.timeout = 600;
   set_port(com2, &portset);
```

```
    release_resource(IO_SYSTEM);

    // start second polling task example
    create_task(master2, 100, applicationGroup, 4);

    // define master message to read slave
    // analog inputs
    message.stream         = com2;
    message.function       = 4;
    message.slaveStation   = 2;
    message.slaveRegister  = 30001;
    message.masterRegister = 40001;
    message.length         = 8;
    message.timeout        = 30;
    message.eventRequest   = FALSE;
    message.eventNo        = 0;

    // main communication loop
    while (TRUE)
    {
        // send a new command
        request_resource(IO_SYSTEM);
        status = serialModbusMaster(&message, &masterSession1);
        release_resource(IO_SYSTEM);

        if(status)
        {
            // wait for response or timeout
            while(masterSession1.masterCmdStatus == MM_SENT)
            {
                // do other things here...

                // allow other tasks to execute while waiting
                release_processor();
            }

            if(masterSession1.masterCmdStatus == MM_RECEIVED)
            {
                good++;
            }
            else
            {
                bad++;
            }
        }

        // allow other tasks to execute
        release_processor();
    }
}

/* ------------------------------------------
   master2

   This task sends commands to a Modbus slave
   using the same serial port as main(). Use a
   different MODBUS_SESSION structure when
   sharing a serial port with another master.

   This task uses the event request option. The
   task waits for the completion event to free
   up the processor for other tasks.
```

```
   ------------------------------------------ */
static void master2(void)
{
   MASTER_MESSAGE   message;
   BOOLEAN          status;
   UINT16           good, bad;

   // define master message to copy slave
   // digital inputs to master outputs
   message.stream         = com2;
   message.function    = 2;
   message.slaveStation   = 2;
   message.slaveRegister  = 10001;
   message.masterRegister = 1;
   message.length         = 8;
   message.timeout        = 30;
   message.eventRequest   = TRUE;
   message.eventNo        = 1;

   // main communication loop
   while (TRUE)
   {
      // send a new command
      request_resource(IO_SYSTEM);
      status = serialModbusMaster(&message, &masterSession2);
      release_resource(IO_SYSTEM);

      if(status)
      {
         // wait for completion event when response or
         // timeout has occurred
         wait_event(1);

         if(masterSession2.masterCmdStatus == MM_RECEIVED)
         {
            good++;
         }
         else
         {
            bad++;
         }
      }

      // allow other tasks to execute
      release_processor();
   }
}
```

Master Message Example Using mTcpMasterMessage

This program sends master messages on the LAN interface using Modbus/TCP protocol.

```
/* ------------------------------------------
   SCADAPack2 C++ Application Main Program
   Copyright 2001 - 2004, Control Microsystems Inc.
   ------------------------------------------ */
#include <ctools.h>

// master IP modes
typedef enum masterIPModes_t
{
   MIP_OPEN_CONNECTION = 0,
   MIP_CONNECTING,
   MIP_SEND_MESSAGE,
   MIP_WAIT_FOR_RESPONSE,
   MIP_DISCONNECT,
   MIP_CLOSE
}
MIP_MODE;

/* ------------------------------------------
   main

   This routine is the main application loop.
   ------------------------------------------ */
int main(void)
{
   MIP_MODE            mode;
   IP_SETTINGS         ipSettings;
   IP_ADDRESS          remoteIP;
   IP_PROTOCOL_TYPE    protocolType;
   CONNECTION_TYPE     appType;
   UINT16              timeout;
   UINT32              connectID;
   MODBUS_CMD_STATUS   cmdStatus;
   BOOLEAN             status;
   UINT16              function;
   UINT16              slaveStation;
   UINT16              slaveRegister;
   UINT16              masterRegister;
   UINT16              length;

   // IP settings for SCADAPack LAN interface
   ipSettings.ipConfigMode = IPConfig_GatewayOnLAN;
   ipSettings.ipAddress[0] = inet_addr("172.16.10.0");
   ipSettings.gateway[0]   = inet_addr("172.16.0.1");
   ipSettings.netMask      = inet_addr("255.255.0.0");
   ipSettings.ipVersion    = 4;
   status = ethernetSetIP(&ipSettings);

   // master IP command definition
   remoteIP.s_addr  = inet_addr("172.16.3.8"); // destination IP address
   protocolType     = IPP_ModbusTcp;
   appType          = CT_MasterCApp;
   timeout          = 30;                      // tenths of seconds
   function         = 3;                       // read holding registers
   slaveStation     = 1;
   slaveRegister    = 40155;
   masterRegister   = 40001;
   length           = 2;
```

```
// main loop
mode = MIP_OPEN_CONNECTION;
while (TRUE)
{
    switch(mode)
    {
        case MIP_OPEN_CONNECTION:
        {
            // open a connection
            status = mTcpMasterOpen(
                remoteIP,
                protocolType,
                appType,
                timeout,
                &connectID,
                &cmdStatus
                );
            if (status)
            {
                mode = MIP_CONNECTING;
            }
        }
        break;

        case MIP_CONNECTING:
        {
            // check master command status
            status = mTcpMasterStatus(connectID, &cmdStatus);
            if (status)
            {
                switch (cmdStatus)
                {
                case MM_CONNECTING:
                    break;
                case MM_CONNECTED:
                    mode = MIP_SEND_MESSAGE;
                    break;
                default:
                    // remaining status codes are error codes
                    mode = MIP_DISCONNECT;
                    break;
                }
            }
        }
        break;

        case MIP_SEND_MESSAGE:
        {
            // send master IP message
            cmdStatus = mTcpMasterMessage(
                connectID,
                remoteIP,
                protocolType,
                function,
                slaveStation,
                slaveRegister,
                masterRegister,
                length,
                timeout
                );

            switch (cmdStatus)
```

```
            {
             case MM_CONNECTING:
             case MM_DISCONNECTING:
             case MM_DISCONNECTED:
                // last command is still being sent;
                // not ready for new message
                break;
            case MM_SENT:
                // message send started successfully
                mode = MIP_WAIT_FOR_RESPONSE;
                break;
            default:
                // remaining status codes are error codes
                // message not sent
                mode = MIP_DISCONNECT;
                break;
            }
        }
        break;

        case MIP_WAIT_FOR_RESPONSE:
        {
            // check master command status
            status = mTcpMasterStatus(connectID, &cmdStatus);
            if (status)
            {
                switch (cmdStatus)
                {
                case MM_SENT:
                    // still waiting for response
                    break;
                case MM_RECEIVED:
                    // response received successfully; send next message
                    mode = MIP_SEND_MESSAGE;
                    break;
                default:
                    // remaining status codes are error codes
                    mode = MIP_DISCONNECT;
                    break;
                }
            }
        }
        break;

        case MIP_DISCONNECT:
        if (mTcpMasterDisconnect(connectID))
        {
            // disconnect is successfully started
            mode = MIP_CLOSE;
        }
        break;

        case MIP_CLOSE:
        if (mTcpMasterClose(connectID))
        {
            // connection has been successfully released
            // open new connection and start again
            mode = MIP_OPEN_CONNECTION;
        }
        break;
    }

    // release processor to other priority 254 tasks
```

```
            release_processor();
        }
    }
```

Modem Initialization Example

The following code shows how to initialize a modem. Typically, the modem initialization is used to prepare a modem to answer calls. The example sets up a Hayes modem to answer incoming calls.

```
#include <ctools.h>
#include <string.h>

int main(void)
{
   struct ModemInit initSettings;
   reserve_id portID;
   enum DialError status;
   enum DialState state;
   struct pconfig portSettings;

   /* Configure serial port 1 */
   portSettings.baud      = BAUD1200;
   portSettings.duplex    = FULL;
   portSettings.parity    = NONE;
   portSettings.data_bits = DATA8;
   portSettings.stop_bits = STOP1;
   portSettings.flow_rx   = RFC_MODBUS_RTU;
   portSettings.flow_tx   = TFC_NONE;
   portSettings.type      = RS232_MODEM;
   portSettings.timeout   = 600;
   request_resource(IO_SYSTEM);
   set_port(com1, &portSettings);
   release_resource(IO_SYSTEM);

   /* Initialize Hayes modem to answer incoming calls */
   initSettings.port = com1;
   strcpy(initSettings.modemCommand, " F1Q0V1X1 S0=1");
   if (modemInit(&initSettings, &portID) == DE_NoError)
   {
      do
      {
         /* Allow other tasks to execute */
         release_processor();

         /* Wait for the initialization to complete */
         modemInitStatus(com1, portID, &status, &state);
      }
      while (state == DS_Calling);

      /* Terminate the initialization */
      modemInitEnd(com1, portID, &status);
   }
}
```

Real Time Clock Program Example

The following program illustrates how the date and time can be set and displayed. All fields of the clock structure must be set with valid values for the clock to operate properly.

```
#include <ctools.h>

int main(void)
{
   TIME now;

   /* Set to 12:01:00 on January 1, 1997 */

   now.hour      = 12;    /* set the time */
   now.minute    = 1;
   now.second    = 0;
   now.day       = 1;     /* set the date */
   now.month     = 1;
   now.year      = 97;
   now.dayofweek = 3;      /* day is Wed. */

   request_resource(IO_SYSTEM);
   setclock(&now);

   getclock(&now);        /* read the clock */
   release_resource(IO_SYSTEM);

   /* Display current hour, minute and second */

   fprintf(com1,"%2d/%2d/%2d", now.day, now.month, now.year);
   fprintf(com1,"%2d:%2d\r\n",now.hour, now.minute);
}
```

Start Timed Event Example

This program prints the time every 10 seconds.

```
#include <string.h>
#include <ctools.h>

#define TIME_TO_PRINT  15

/* ------------------------------------------
   The shutdown function stops the signalling
   of TIME_TO_PRINT events.
   ------------------------------------------ */
void shutdown(void)
{
   endTimedEvent(TIME_TO_PRINT);
}

/* ------------------------------------------
   The main function sets up signalling of
   a timed event, then waits for that event.
   The time is printed each time the event
   occurs.
   ------------------------------------------ */
int main(void)
{
   struct prot_settings settings;
   struct clock now;
   TASKINFO taskStatus;

   /* Disable the protocol on serial port 1 */
   settings.type = NO_PROTOCOL;
   settings.station = 1;
   settings.priority = 250;
   settings.SFMessaging = FALSE;
   request_resource(IO_SYSTEM);
   set_protocol(com1, &settings);
   release_resource(IO_SYSTEM);

   /* set up task exit handler to stop
      signalling of events when this task ends */
   getTaskInfo(0, &taskStatus);
   installExitHandler(taskStatus.taskID, (FUNCPTR) shutdown);

   /* start timed event */
   startTimedEvent(TIME_TO_PRINT, 100);

   while (TRUE)
   {
      wait_event(TIME_TO_PRINT);
      request_resource(IO_SYSTEM);
      getclock(&now);
      release_resource(IO_SYSTEM);
      fprintf(com1, "Time %02u:%02u:%02u\r\n", now.hour,
      now.minute, now.second);
   }
}
```

Porting Existing C Tools Applications

Porting SCADAPack 32 C++ Applications to the SCADAPack2

Compiler Differences between Hitachi and GNU

The Hitachi compiler used with the SCADAPack 32 has the following difference with GNU compiler used with the SCADAPack2:

The order of bit fields is reversed. Bit field ordering is not specified by the C standard. It is left to the compiler maker. Existing programs using bit fields must be modified if the order of the bit fields affects the operation of the program. If the bit fields are being used only for space efficiency the program does not need rewriting.

Porting Existing C++ Tools Applications

Existing SCADAPack 32 C++ applications are highly compatible with the SCADAPack2 C++ Tools. However changes are necessary. The following guide describes the steps in porting an application.

Copy SCADAPack2 C++ Application Framework

Begin by making a copy of the SCADAPack2 C++ application framework using the ISaGRAF sample application or the TelePACE sample application. By default the samples are installed at `C:\SP2 Applications`. Make a copy of one of the following directories:

```
C:\SP2 Applications\ISaGRAF\Sample
C:\SP2 Applications\TelePACE\Sample
```

Changes to appstart.cpp

Instead of `appSettings.src` used in SCADAPack 32 C++ applications, the new `appstart.cpp` assigns the stack size as well as the main task priority. Task priorities are discussed under changes to the function **`create_task`**. The heap size is no longer configurable. The C++ application has access to the entire system heap.

Open the sample `appstart.cpp` to review these application settings:

```
...
// Priority of the task main().
// Priority 100 is recommended for a continuously running task.
UINT32 mainPriority = 100;

// Stack space allocated to the task main().
// Note that at least 5 stack blocks are needed to call fprintf().
UINT32 mainStack = 5;

// Application group assigned to the task main().
// A unique value is assigned by the system to the applicationGroup
// for this application. Use this variable in calls to create_task()
```

```
// by this application.
UINT32 applicationGroup = 0;
...
```

A C++ application should not require any further modifications to `appstart.cpp`. Note that there are no longer function calls in `appstart()` for starting various drivers as there were in the SCADAPack 32 version. These drivers are already running when a C++ Application is executed. It is still possible to call these functions to disable functionality. For example, **`runTarget`**`(FALSE)` may still be called from `appstart()` or `main()` to stop the logic application.

Replace main.cpp

Replace the sample file `main.cpp` with the `main.cpp` from your SCADAPack 32 C++ Application. Edit your `main.cpp` and make the following changes:

1. In addition to the `ctools.h` header you must include the header file `nvMemory.h`.
2. The C++ Tools require `main()` to have the prototype: `int main(void)`. Change the syntax of `main()` so that it returns the data type `int` instead of `void`. Note that the returned `int` value is not accessible to the user and so any value may be returned or none at all.
3. Remove the function definition for `abort()`. This function is provided by the operating system.
4. The call `release_processor()` in the main loop can be deleted. See section **Operating System Scheduling** for details.

Add Remaining C and CPP Files

Copy any additional C, CPP or H files from your application to the copied sample application directory.

Replace Partially Supported and Unsupported Functions

Existing programs may use some functions that are partially supported or unsupported on the SCADAPack2 controller. The program must be changed to use the new functions of the SCADAPack2 controller. For a list of the functions affected refer to the sections **Partially Supported C++ Tools Functions**,.

Build the Application

The SCADAPack2 C++ Tools use a DOS command line to compile and link a C++ application. The sample application includes the command file `build.bat` to do this. Please see the section ***Application Development*** for more details on editing `build.bat,` command line options, and loading the application into the controller.

Test the Application

This step is specific to the application. It must be tested to confirm it operates correctly on the SCADAPack2 controller. You also should pay attention to the following.

- SCADAPack 32 controllers have higher performance than do SCADAPack2 controllers. Check that any I/O operations allow enough time for field signals to change state. Some timing relationships in the existing program may not be true in the new program, depending on how you have implemented them. For example, a calculation done between two I/O operations may execute slower and cause the second I/O operation to take place later than you want.

- Check that any periodic functions execute at the correct rate. If you've used standard timing functions this should not be a problem. If you've used delay loops then these will execute slower. You should replace them with standard timing functions.

Partially Supported C++ Tools Functions

The following sections describe functions that are supported by the SCADAPack 32 C++ Tools but are only partially supported by the SCADAPack2 C++ Tools. The following features are similar to existing SCADAPack 32 C++ Tools features but require some source code modification.

Refer to these sections when porting existing SCADAPack 32 C++ Tools Applications to the SCADAPack2.

Event Numbers for SCADAPack2 C++ Applications

The SCADAPack2 supports up to 32 separate user-loaded C++ Applications. Event numbers 0 to 31 were made available to the SCADAPack 32 C++ application. This same event number range must be shared on the SCADAPack2 among the user-loaded C++ Applications.

The RealFLO C++ Application uses events 20, 21, or 22. These events may not be used by other C++ Applications when the RealFLO C++ Application is loaded in the SCADAPack2.

Stack used by fprintf Function

Tasks that call the function **`fprintf`** require at least 5 stack blocks. This function required only 4 stack blocks when used in SCADAPack 32 C++ applications. As a general rule, add 1 stack block to the amount used in a SCADAPack 32 C++ application.

Macro stdout is Disabled

The macro **stdout** is disabled in the SCADAPack2 C++ Tools. Instead use the serial port macros: com1, com2, or com3. This means that the following functions that use stdout do not work: **`printf`**, **`putc`**, **`getc`**. Use the replacement functions listed below.

Function	Replaced with
`printf`	`fprintf`
`putc`	`fputc`
`getc`	`fgetc`

Task Creation Function

The task priorities have changed with the SCADAPack2. There are now 255 priority levels, and the highest priority task has a priority of 0. Existing calls to **`create_task`** will need to be modified to account for a lower number being a higher priority.

The table below contains the recommended priority values to use when porting to the SCADAPack2.

Priority Description	Equivalent Priority Value for SCADAPack2	Priority Value for SCADAPack 32
Higher Priority	25	4
	50	3
	75	2
Lower Priority	100	1

The argument used for application type in existing calls to `create_task` must be replaced with the global variable `applicationGroup`. The operating system assigns a unique value to `applicationGroup` when it is defined in appStart.cpp.

Please see the documentation for `create_task` in the ***Function Specifications*** section for more details.

Controller I/O Functions

The following functions are no longer supported. The replacement function is indicated for each. Each function is documented in the ***Function Specifications*** section.

Function	Replaced with
`interruptInput`	`no replacement function`
`interruptCounter`	`no replacement function`
`readCounter`	`ioReadCounterSP2`
`readCounterInput`	`no replacement function`
`ioReadDin5232`	`no replacement function`
`ioReadCounter5232`	`ioReadCounterSP2`
`ioRead5601Inputs`	`ioReadSP2Inputs`
`ioRead5601Outputs`	`ioReadSP2Outputs`
`ioWrite5601Outputs`	`ioWriteSP2Outputs`

Exit Handler Function

The argument used to specify the exit handler function in existing calls to **`installExitHandler`** must be cast to the type `(FUNCPTR)`. Please see the documentation for `installExitHandler` in the ***Function Specifications*** section for more details.

Program Status Functions

The functions **`getProgramStatus`** and **`setProgramStatus`** have changed syntax. To support multiple C++ applications, these functions now have an argument to specify the application. The new syntax for these functions is documented in the ***Function Specifications*** section.

Freeing Dynamic Memory

When a C++ Application is ended (e.g. by using the *STOP* button from the C/C++ Program Loader), memory allocated by using the **malloc** function is not freed automatically. An exit handler must be installed to free allocated memory. Please see the documentation for `installExitHandler` in the ***Function Specifications*** section for more details.

Non-Volatile Data Sections

The SCADAPack2 has a different method for declaring static non-volatile memory. There is still 8 kB of memory available but it must now be shared with all user-loaded C++ applications. Non-volatile variable declarations and their pragma statements must be removed from each source file and declared globally in the one file `nvMemory.h`. Include `nvMemory.h` in each source file that uses non-volatile variables.

Please see the section ***Non-Volatile Memory*** for more details on editing `nvMemory.h` and on using these variables in your source files.

Socket Functions

The following functions are no longer supported. The replacement function is indicated for each.

Function	Replaced with
tfClose	close
tfGetSocketError	errnoGet

Modbus Handler Functions

The **installModbusHandler** is used to add user-defined extensions to the standard Modbus protocol. To uninstall a Modbus handler in a SCADAPack 32 C++ application, the same function is called with the NULL pointer.

SCADAPack2 C++ applications support the installation of multiple Modbus handlers. In order to remove a specific Modbus handler the new function **removeModbusHandler** is used. Note that calling installModbusHandler with the NULL pointer has no effect.

Unsupported C++ Tools Functions

The following sections describe functions that are supported by the SCADAPack 32 C++ Tools but are not supported by the SCADAPack2 C++ Tools.

Refer to these sections when porting existing C++ Tools Applications to the SCADAPack2.

Timers

The following C++ Tools Timer functions are not supported. Use the functions **readStopwatch** or **startTimedEvent** instead.

Function
interval
read_timer_info
runTimers
settimer
timer

Option Switch Function

The following C++ Tools function is not supported.

Function
optionSwitch

IP Functions

The following C++ Tools functions are not supported.

Function
readv
tfBcopy
tfBindNoCheck
tfBlockingState
tfBzero
tfDialerAddExpectSend
tfDialerAddSendExpect
tfFreeZeroCopyBuffer
tfGetOobDataOffset
tfGetPppDnsIpAddress

Function
tfGetPppPeerIpAddress
tfGetSendCompltBytes
tfGetWaitingBytes
tfGetZeroCopyBuffer
tfInetToAscii
tfIoctl
tfPingClose
tfPingGetStatistics
tfPingOpenStart
tfPppSetOption
tfRead
tfRegisterSocketCB
tfRegisterSocketCBParam
tfResetConnection
tfSendToInterface
tfSetPppPeerIpAddress
tfSetTreckOptions
tfSocketArrayWalk
tfUseDialer
tfWrite
tfZeroCopyRecv
tfZeroCopyRecvFrom
tfZeroCopySend
tfZeroCopySendTo
writev

PPP Functions

The following C++ Tools PPP functions are not supported.

Function
pppGetInterfaceHandle
pppReadSettings
pppReadUserTableEntry
pppReadUserTableSize
pppWriteSettings
pppWriteUserTableEntry
pppWriteUserTableSize

Porting SCADAPack C Applications to the SCADAPack2

Porting Existing C Tools Applications

Existing SCADAPack C applications are highly compatible with the SCADAPack2 C++ Tools. However changes are necessary. The following guide describes the steps in porting an application.

Copy SCADAPack2 C++ Application Framework

Begin by making a copy of the SCADAPack2 C++ application framework using the ISaGRAF sample application or the TelePACE sample application. By default the samples are installed at `C:\SP2 Applications`. Make a copy of one of the following directories:

`C:\SP2 Applications\ISaGRAF\Sample`

```
C:\SP2 Applications\TelePACE\Sample
```

Changes to appstart.cpp

The new `appstart.cpp` assigns the stack size as well as the main task priority. Task priorities are discussed under changes to the function **`create_task`**. The heap size is no longer configurable. The C++ application has access to the entire system heap.

Open the sample `appstart.cpp` to review these application settings:

```
...
// Priority of the task main().
// Priority 100 is recommended for a continuously running task.
UINT32 mainPriority = 100;

// Stack space allocated to the task main().
// Note that at least 5 stack blocks are needed to call fprintf().
UINT32 mainStack = 5;

// Application group assigned to the task main().
// A unique value is assigned by the system to the applicationGroup
// for this application. Use this variable in calls to create_task()
// by this application.
UINT32 applicationGroup = 0;
...
```

A C++ application should not require any further modifications to `appstart.cpp`. Note that there are no longer function calls in `appstart()` for starting various drivers as there were in the SCADAPack version. These drivers are already running when a C++ Application is executed. It is still possible to call these functions to disable functionality. For example, **`runTarget`**`(FALSE)` may still be called from `appstart()` or `main()` to stop the logic application.

Add Existing Program Files to Framework

1. Copy all user-written *.C files from the SCADAPack application to the SCADAPack2 framework directory created in the last section.
2. Copy user-written *.H files, if any, from the SCADAPack application to the framework directory. Do NOT copy the SCADAPack ctools.h file or any other C Tools header files (e.g. older SCADAPack C Tools headers such as protocol.h). The SCADAPack2 ctools.h is already in the framework directory.
3. For each user-written *.H file copied to the framework directory in step 2, make sure that the following statements are included at the top of each header file:

   ```
   #ifdef __cplusplus
      extern "C"
   {
   #endif
   ```

   And also make sure that the following statements are included at the bottom of each header file:

   ```
   #ifdef __cplusplus
   }
   #endif
   ```

4. Edit the SCADAPack application file that contains the function `main()`. Open this file and copy its contents beginning after the included headers and paste this into the framework file main.cpp after the prototypes as shown below. If there are additional headers included, copy these include statements to the main.cpp file next.

```
/* ------------------------------------------------------------
   SCADAPack2 C++ Application Main Program
   Copyright 2006, Control Microsystems Inc.
   ------------------------------------------------------------ */
#include <ctools.h>
#include "nvMemory.h"

/* ------------------------------------------------------------
   C++ Function Prototypes
   ------------------------------------------------------------ */
// add prototypes here

/* ------------------------------------------------------------
   C Function Prototypes
   ------------------------------------------------------------ */
extern "C"
{
   // add prototypes here
}

Paste your code here
```

5. Delete the additional stub function main() at the end of the file main.cpp. The C++ Tools require `main()` to have the prototype: `int main(void)`. Change the syntax of `main()` so that it returns the data type `int` instead of `void`. Note that the returned `int` value is not accessible to the user and so any value may be returned or none at all.

Replace Older C Tools Headers with ctools.h

If the ported application used SCADAPack C Tools version 2.12 or older, the program C files will likely have include statements with C Tools header files, such as protocol.h, primitiv.h, etc. Replace all these C Tools include statements in all program C files with just one include statement:

```
include <ctools.h>
```

Replace Partially Supported and Unsupported Functions

Existing programs may use some functions that are partially supported or unsupported on the SCADAPack2 controller. The program must be changed to use the new functions of the SCADAPack2 controller. For a list of the functions affected refer to the sections **Partially Supported C Tools Functions**.

Build the Application

The SCADAPack2 C++ Tools use a DOS command line to compile and link a C++ application. The sample application includes the command file `build.bat` to do this. Please see the section ***Application Development*** for more details on editing `build.bat,` command line options, and loading the application into the controller.

Test the Application

This step is specific to the application. It must be tested to confirm it operates correctly on the SCADAPack2 controller. You also should pay attention to the following.

- SCADAPack2 controllers have higher performance than do SCADAPack controllers. Check that any I/O operations allow enough time for field signals to change state. Some timing relationships in the existing program may not be true in the new program, depending on how you have implemented them. For example, a calculation

done between two I/O operations may execute faster and cause the second I/O operation to take place sooner than you want.

- Check that any periodic functions execute at the correct rate. If you've used standard timing functions this should not be a problem. If you've used delay loops then these will execute faster. You should replace them with standard timing functions.

Partially Supported C Tools Functions

The following sections describe functions that are supported by the TelePACE C Tools and ISaGRAF C Tools but are only partially supported by the SCADAPack2 C++ Tools. The following features are similar to existing C Tools features but require some source code modification.

Refer to these sections when porting existing SCADAPack C Tools Applications to the SCADAPack2.

Event Numbers for SCADAPack2 C++ Applications

The SCADAPack2 supports up to 32 separate user-loaded C++ Applications. Event numbers 0 to 31 were made available to the SCADAPack C application. This same event number range must be shared on the SCADAPack2 among the user-loaded C++ Applications.

The RealFLO C++ Application uses events 20, 21, or 22. These events may not be used by other C++ Applications when the RealFLO C++ Application is loaded in the SCADAPack2.

Stack used by fprintf Function

Tasks that call the function **`fprintf`** require at least 5 stack blocks. This function required only 4 stack blocks when used in SCADAPack C applications. As a general rule, add 1 stack block to the amount used in a SCADAPack application.

Macro stdout is Disabled

The macro **stdout** is disabled in the SCADAPack2 C++ Tools. Instead use the serial port macros: com1, com2, or com3. This means that the following functions that use stdout do not work: **`printf`**, **`putc`**, **`getc`**. Use the replacement functions listed below.

Function	Replaced with
`printf`	`fprintf`
`putc`	`fputc`
`getc`	`fgetc`

Task Creation Function

The task priorities have changed with the SCADAPack2. There are now 255 priority levels, and the highest priority task has a priority of 0. Existing calls to **`create_task`** will need to be modified to account for a lower number being a higher priority.

The table below contains the recommended priority values to use when porting to the SCADAPack2.

Priority Description	Equivalent Priority Value for SCADAPack2	Priority Value for SCADAPack
`Higher Priority`	25	4
	50	3
	75	2

Priority Description	Equivalent Priority Value for SCADAPack2	Priority Value for SCADAPack
Lower Priority	100	1

The argument used for application type in existing calls to `create_task` must be replaced with the global variable `applicationGroup`. The operating system assigns a unique value to `applicationGroup` when it is defined in appStart.cpp.

Please see the documentation for `create_task` in the ***Function Specifications*** section for more details.

Exit Handler Function

The argument used to specify the exit handler function in existing calls to **`installExitHandler`** must be cast to the type `(FUNCPTR)`. Please see the documentation for `installExitHandler` in the ***Function Specifications*** section for more details.

Program Status Functions

The functions **`getProgramStatus`** and **`setProgramStatus`** have changed syntax. To support multiple C++ applications, these functions now have an argument to specify the application. The new syntax for these functions is documented in the ***Function Specifications*** section.

Freeing Dynamic Memory

When a C++ Application is ended (e.g. by using the *STOP* button from the C/C++ Program Loader), memory allocated by using the **malloc** function is not freed automatically. An exit handler must be installed to free allocated memory. Please see the documentation for `installExitHandler` in the ***Function Specifications*** section for more details.

Non-Volatile Data Sections

C Tools applications could make any variable non-volatile by renaming the memory section where it was located. This was done using a compiler `pragma` directive. This is not supported on the SCADAPack2.

SCADAPack2 C++ Tools applications can make variables non-volatile by locating them in SRAM. There is 8 KB of SRAM available for static application variables. If this is not enough, up to 1 MB of SRAM is available for dynamic non-volatile memory allocation. See the function **allocateMemory**.

To create a non-volatile section, refer to the section ***Non-Volatile Memory (nvMemory.h)*.**

I/O System Functions

The SCADAPack2 and SCADAPack 32 use a different I/O system architecture than the SCADAPack. I/O operations can be performed in parallel with application program execution. This improves the performance of ISaGRAF and TelePACE applications, and can have similar impact on user applications.

In the new architecture, I/O requests are added to a queue. Requests are read from the queue and processed by separate I/O controller hardware. Data are stored in I/O arrays that can be read and written by the application program. The application program can

also synchronize with the I/O controller to determine when a set of I/O requests is complete.

Existing application programs must be rewritten to use the new I/O system functions.

Most I/O System functions are C++ functions. In order to call C++ functions from a source file, the source file must be a *.CPP file. If an existing *.C file must be renamed to a *.CPP file.

The following is a list of the I/O System functions. Each function is documented in the ***Function Specifications*** section.

C++	Function	Description
✓	`ioSetConfiguration`	Set I/O controller configuration
✓	`ioGetConfiguration`	Get I/O controller configuration
✓	`ioVersion`	Get I/O controller firmware version
	`ioNotification`	Request notification
	`ioSystemReset`	Request reset of all I/O modules
	`ioRequest`	Request I/O module scan
	`ioStatus`	Read I/O module status
✓	`ioReadAin4`	Read buffered data from 4 point analog input module
✓	`ioReadAin8`	Read buffered data from 8 point analog input module
✓	`ioReadAout2`	Read buffered data for 2 point analog output module
✓	`ioReadAout4`	Read buffered data for 4 point analog output module
✓	`ioReadCounter4`	Read buffered data from 4 point counter input module
✓	`ioReadCounterSP2`	Read buffered data from SCADAPack2 counters
✓	`ioReadDin16`	Read buffered data from 16 point digital input module
✓	`ioReadDin8`	Read buffered data from 8 point digital input module
✓	`ioReadDout16`	Read buffered data for 16 point digital output module
✓	`ioReadDout8`	Read buffered data for 8 point digital output module
✓	`ioReadSP2Inputs`	Read buffered data from SCADAPack2 inputs
✓	`ioReadSP2Outputs`	Read buffered data for SCADAPack2 outputs
✓	`ioWriteAout2`	Write buffered data for 2 point analog output module
✓	`ioWriteAout4`	Write buffered data for 4 point analog output module
	`ioWriteDout16`	Write buffered data for 16 point digital output module
	`ioWriteDout8`	Write buffered data for 8 point digital output module
	`ioWriteSP2Outputs`	Write buffered data for SCADAPack2 outputs

Controller I/O Functions

The following functions are no longer supported. The replacement function is indicated for each.

Function	Replaced with
interruptInput	no replacement function
interruptCounter	no replacement function
readCounter	ioReadCounterSP2
readCounterInput	no replacement function
readInternalAD	readBattery, readThermistor
ioReset	ioSystemReset
ioRefresh	functions in the ioWrite group
ioReadDin5232	no replacement function
ioReadCounter5232	ioReadCounterSP2
ioRead5601Inputs	ioReadSP2Inputs
ioRead5601Outputs	ioReadSP2Outputs
ioWrite5601Outputs	ioWriteSP2Outputs

ISaGRAF I/O Functions

The I/O System functions are used in place of the following ISaGRAF C++ Tools I/O functions which are no longer supported.

Function	Replaced with
isaRead4Ain	ioReadAin4
isaRead8Ain	ioReadAin8
isaRead4Counter	ioReadCounter4
isaRead8Din	ioReadDin8
isaRead16Din	ioReadDin16
isaRead5601Inputs	ioReadSP2Inputs
isaWrite2Aout	ioWriteAout2
isaWrite4Aout	ioWriteAout4
isaWrite8Dout	ioWriteDout8
isaWrite16Dout	ioWriteDout16
isaWrite5601Outputs	ioWriteSP2Outputs

Backwards Compatibility I/O Functions

The following I/O related functions were available in the original release of the TelePACE C++ Tools. They were supported for backward compatibility in later versions of the TelePACE C++ Tools, but did not allow access to all I/O modules. They are no longer compatible with the I/O system architecture.

These functions are replaced with equivalent I/O system functions. The new functions provide access to all I/O modules.

Function	Replaced with
dout	functions in the ioWrite group
din	functions in the ioRead group
aout	functions in the ioWrite group
ain	functions in the ioRead group

Other I/O Function Changes

The following C++ Tools I/O functions are fully supported in the SCADAPack2 C++ Tools with the following difference. Instead of executing the required I/O operation immediately before returning from the function, the I/O operation is added to the I/O System queue as an I/O request and is performed by the I/O System architecture in parallel with application program execution.

Notification of the completion of an I/O request may be obtained using the **ioNotification** function.

Function	Description
hartIO	Request a hart I/O module scan. The scan reads data from the 5904 interface module, processes HART responses, processes HART commands, and writes commands and configuration data to the 5904 interface module.
ioClear	Request all I/O points to be cleared.

Jiffy Clock Functions

The C Tools function `jiffy` is replaced with the **`readStopwatch`** function. This function returns the time in milliseconds. Existing programs must be modified to call the new function and to convert any timing constants to milliseconds.

The C Tools function `setjiffy` is not supported. Elapsed time from a particular point can be measured by saving the time at the start of the interval, rather than setting the clock to zero.

Real Time Clock Functions

The **`getclock`** function has a new syntax. A clock structure is no longer returned by the function. Instead a pointer to a clock structure is passed as an argument. The `getclock` function is documented in the ***Function Specifications*** section.

Get Task Information Function

The **`getTaskInfo`** function has a new syntax. A TASKINFO structure is no longer returned by the function. Instead a pointer to a TASKINFO structure is passed as an argument and a status flag is returned. The `getTaskInfo` function is documented in the ***Function Specifications*** section.

EEPROM/Flash Memory Functions

SCADAPack2 controllers use flash memory instead of EEPROM to store controller settings. The following functions are no longer supported. The replacement function is indicated for each.

Function	Replaced with
load	flashSettingsLoad
save	flashSettingsSave

Controller Status Function

The controller status functions `setStatusBit` and `getStatusBit` are fully supported. The **`setStatus`** function is no longer supported. Use `setStatusBit` in place of `setStatus`.

Store and Forward Functions

The syntax for the following two functions has been changed. Instead of passing or returning a SFTranslation structure, the new functions pass a pointer to a SF_TRANSLATION structure. See the new function syntax in the sections following.

Function	Description
getSFTranslation	Read Store and Forward Translation
setSFTranslation	Write store and forward translation table entry.

The previous structure, `struct SFTranslation`, shown below is no longer supported.

```
struct SFTranslation {
       unsigned portA;
       unsigned stationA;
       unsigned portB;
       unsigned stationB;
       };
```

This structure is replaced with the structure, SF_TRANSLATION, which includes an IP address field to accommodate store and forward involving the Ethernet port. The structure is defined as:

```
typedef struct st_SFTranslationMTcp
{
       COM_INTERFACE  slaveInterface;    // slave interface type
       UINT16         slaveStation;      // slave station address
       COM_INTERFACE  forwardInterface;  // forwarding interface type
       UINT16         forwardStation;    // forwarding station address
       IP_ADDRESS     forwardIPAddress;  // forwarding IP address
}
SF_TRANSLATION;
```

The following table explains how to correct existing programs that use the older structure. The new SF_TRANSLATION structure is documented following this table.

Item to be replaced	Replacement
struct SFTranslation	The new structure has the macro name SF_TRANSLATION.
portA field	Set *slaveInterface* field = *portA + 1* (1 = com1, 2 = com2, 3 = com3, 100 = Ethernet1)
portB field	Set *forwardInterface* field = *portB + 1* (1 = com1, 2 = com2, 3 = com3, 100 = Ethernet1)
stationA field	*slaveStation* field
stationB field	*forwardStation* field

Instead of entering a translation in any order for the communication interfaces (as done with the old structure), the translation data is entered specifying the receiving slave interface (*slaveInterface* and *slaveStation*) and the forwarding master interface (*forwardInterface*, *forwardStation* and *forwardIPAddress*, if applicable).

Translations describe the communication path of the master command: e.g. the slave interface which receives the command and the forwarding interface to forward the command. The response to the command is automatically returned to the master through the same communication path in reverse.

The getSFTranslation and setSFTranslation functions are documented in the ***Function Specifications*** section.

Serial Port Configuration Functions

portConfiguration

The C Tools function *portConfiguration* returned a pointer to the port configuration table for com1 and com2 only. These functions are no longer supported.

Use the functions *get_port* and *set_port* in place of *portConfiguration.*

Default Settings for Com1 and Com2

The default settings for Com1 and Com2 have changed. All three serial ports of the SCADAPack2 have the same default settings and the same range of available settings. This change is most notable in the default setting for Rx Flow control as described below.

The documentation for the structure *pconfig* has been updated below to reflect these changes.

Rx Flow Control

The C Tools required the Rx Flow Control for com1 and com2 to be set to DISABLE when the Modbus RTU protocol is used. The ports com1 and com2 on the SCADAPack2 must have Rx Flow Control set to RFC_MODBUS_RTU (or ENABLE) when the Modbus RTU protocol is used. Rx Flow Control must be set to RFC_NONE (or DISABLE) when the Modbus ASCII or any other protocol is used.

Rx and Tx Flow Control requirements are now the same for all three serial ports of the SCADAPack2.

New Flow Control MACROS

To help clarify the type of Flow Control feature provided when ENABLE or DISABLE is specified, four new macros have been defined:

RFC_MODBUS_RTU may be used in place of ENABLE. Both have the value 1.

RFC_NONE may be used in place of DISABLE. Both have the value 0.

TFC_IGNORE_CTS may be used in place of ENABLE. Both have the value 1.

TFC_NONE may be used in place of DISABLE. Both have the value 0.

Timeout Setting Not Supported

The timeout serial port setting is no longer supported for com1 and com2. The serial port timeout setting was never supported for com3 or com4 on the SCADAPack controller. This setting is ignored and is fixed at 600ms for backwards compatibility.

Timed Events

Periodic timing may be desired when a continuous loop needs to be repeated at a fixed interval of time. The timed event feature sets up a periodic event that is signaled by the operating system at a specified fixed interval.

A main application task or an additional application task can be made to wait on a periodic event before executing a set of actions. If the actions are completed before the next periodic event has been signaled, the task is blocked while waiting for the event. This blocked state allows the processor to execute other application or system tasks while it waits. This is more efficient than executing a loop that checks for a timer to expire.

For an example using timed events see the function **startTimedEvent**.

Reading the System Stopwatch

For one-time actions and timed actions that need accuracy better that a tenth of a second, the system clock may be read using the function **readStopwatch**. This function returns the system time in milliseconds and has a resolution of 10 ms. The stopwatch time rolls over to 0 when it reaches the maximum value for an unsigned long int (i.e. a UINT32): 4,294,967,295 ms (or about 49.7 days).

For example,

```
startTime = readStopwatch();

// wait for 50 ms (+/- 10 ms)
while ((readStopwatch() – startTimed) < 50)
{
```

```
        release_processor();
}
```

Refer to the section describing the function **readStopwatch** for other timing examples using this function.

Modbus Handler Functions

The **`installModbusHandler`** is used to add user-defined extensions to the standard Modbus protocol. To uninstall a Modbus handler in a SCADAPack C application, the same function is called with the NULL pointer.

SCADAPack2 C++ applications support the installation of multiple Modbus handlers. In order to remove a specific Modbus handler the new function **`removeModbusHandler`** is used. Note that calling `installModbusHandler` with the NULL pointer has no effect.

Unsupported C Tools Functions

The following sections describe functions that are supported by the TelePACE C Tools and ISaGRAF C Tools but are not supported by the SCADAPack2 C++ Tools.

Refer to these sections when porting existing SCADAPack C Tools Applications to the SCADAPack2.

Application Checksum Function

A checksum is no longer used for the C++ application. The C Tools function `applicationChecksum` is not supported.

Backwards Compatibility Functions

These functions were supported in previous C Tools for backwards compatibility, however they were stubs. The following C Tools functions are not supported.

Function
`setSFMapping`
`getSFMapping`

Boot Type Functions

These functions are not useful to C++ Applications. The following C Tools functions are not supported.

Function
`setBootType`
`getBootType`

I/O Bus Communication Functions

The SCADAPack2 I/O System does not support these C Tools functions. These functions provide user access to third party I^2C compatible devices. Without these functions access is limited to Control Microsystems I/O modules only.

Function
`ioBusStart`
`ioBusStop`
`ioBusReadByte`
`ioBusReadLastByte`
`ioBusWriteByte`
`ioBusSelectForRead`

Function
ioBusSelectForWrite
ioBusReadMessage
ioBusWriteMessage

Timers

The following C++ Tools Timer functions are not supported. Use the functions **readStopwatch** or **startTimedEvent** instead.

Function
interval
read_timer_info
settimer
timer